NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DENIAL OF SERVICE ATTACKS ON 802.1X SECURITY PROTOCOL

by

Orhan Ozan

March 2004

Thesis Advisor: Geoffrey Xie
Second Reader: John Gibson

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY	**2. REPORT DATE** March 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Denial of Service Attacks on 802.1X Security Protocol			**5. FUNDING NUMBERS**
6. AUTHOR(S) Orhan OZAN			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Wireless Local Area Networks (WLANs) are quickly becoming popular in daily life. Users are adopting the latest technology to save time and costs. In addition, WLANs are providing high-speed network access to the users. There are security concerns with WLANs that must be considered when deploying them over critical infrastructure, such as military and administrative government LANs.

The IEEE 802.11 wireless standard specifies both an authentication service and encryption protocol, but research has demonstrated that these protocols are severely flawed. The IEEE has established a new workgroup, the IEEE 802.11i, to address all the security vulnerabilities of the 802.11 security protocol. The workgroup proposed using the IEEE 802.1X Port-Based Network Access Control Standard as an interim measure to meet the security requirements of the WLANs and to maintain the confidentiality, authenticity, and availability of the data until the workgroup is finished with the new specifications.

Using an open-source test-bed for evaluating DoS attacks on WLANs, this research demonstrates four different DoS attacks that verify the weaknesses of the IEEE 802.1X protocol. Solutions are provided to mitigate the effects of such DoS attacks.

14. SUBJECT TERMS Wireless Local Area Networks, Security, Denial of Service Attacks, 802.1X Security Protocol, Open Source Test Bed.			**15. NUMBER OF PAGES** 139
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DENIAL OF SERVICE ATTACKS ON 802.1X SECURITY PROTOCOL

Orhan Ozan
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2004

Author: Orhan Ozan

Approved by: Geoffrey Xie, PhD
Thesis Advisor

John Gibson
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Wireless Local Area Networks (WLANs) are quickly becoming popular in daily life. Users are adopting the latest technology to save time and costs. In addition, WLANs are providing high-speed network access to the users. There are security concerns with WLANs that must be considered when deploying them over critical infrastructure, such as military and administrative government LANs.

The IEEE 802.11 wireless standard specifies both an authentication service and encryption protocol, but research has demonstrated that these protocols are severely flawed. The IEEE has established a new workgroup, the IEEE 802.11i, to address all the security vulnerabilities of the 802.11 security protocol. The workgroup proposed using the IEEE 802.1X Port-Based Network Access Control Standard as an interim measure to meet the security requirements of the WLANs and to maintain the confidentiality, authenticity, and availability of the data until the workgroup is finished with the new specifications.

Using an open-source test-bed for evaluating DoS attacks on WLANs, this research demonstrates four different DoS attacks that verify the weaknesses of the IEEE 802.1X protocol. Solutions are provided to mitigate the effects of such DoS attacks.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS

I would like to express my thanks to my thesis advisor Professor Geoffrey Xie and my second reader John Gibson for their help and guidance.

I would also like to thank to my thesis partner Ltjg. Hulusi Onder for his motivation. Without Hulusi's hard work and sincere support, I would never have any chance of completing this thesis.

Finally, I would like to thank to my wife, Hamdiye Banu Ozan for her unconditional love and sacrifice throughout my life.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

Wireless Local Area Networks (WLANs) are quickly becoming part of daily life. Users are adopting the latest technology to save time and costs. In addition, the WLANs are providing high-speed network access to users. The security concerns of WLANs are very important when deployed over critical infrastructure, such as military and government LANs.

The IEEE 802.11 wireless standard specifies both an authentication service and encryption protocol, but research has demonstrated that these protocols are severely flawed. The 802.11 security standard, known as WEP, leaves wireless communications open to several types of attacks. Since WEP has serious flaws, IEEE has established a new workgroup, the IEEE 802.11i, to address all the security vulnerabilities of the 802.11 security protocol.

The workgroup recommended using the IEEE 802.1X Port-Based Network Access Control Standard, as an interim measure, to meet the security requirements of the WLANs and to maintain the confidentiality, authenticity, and availability of the data until a new WLAN security specification is fully developed. At the end of 2003, the workgroup released a draft specification for a Robust Security Network (RSN), but the new protocol has yet to be implemented in commercially available products.

Arunesh Mishra and William Arbaugh from the University of Maryland published a paper claiming that Denial of Service (DoS), Man-in-the-Middle (MiM) and Session Hijacking attacks can be successfully launched against commercial IEEE 802.1X based systems.[12] After this claim, Cisco released a paper as an answer to these claims, indicating MiM and Session Hijacking attacks are not possible on those 802.1X systems fitted with strong data encryption but DoS attacks might be successful as claimed.[13]

In a previous Naval Postgraduate School master's thesis [10], LTJG Selcuk Ozturk, of the Turkish Navy, demonstrated a successful DoS attack against an 802.1X WLAN. The first step of such as attack was to monitor legitimate WLAN communication with Netstumbler, a freeware network monitor, to capture the MAC address and the SSID

information of the legitimate access point. Then a malicious AP is deployed spoofing MAC address and SSID of the valid access point. The malicious AP can be programmed to broadcast deauthentication messages to all the clients. The 802.1X does not require deauthentication messages to be authenticated. Therefore, upon receiving one of the deauthentication messages from the malicious AP, a legitimate client will attempt to re-authenticate with the network. When the attacking AP generates a stronger RF signal than the legitimate AP, the client will try to authenticate with the malicious AP and be denied access to the WLAN.

B. THESIS OBJECTIVES

The main objective of this thesis is to conduct a more systematic evaluation of Denial of Service attacks against WLANs using an open-source test-bed. The thesis will provide a classification framework for all types of DoS attack.

Four different DoS attack scenarios will be presented to demonstrate the weaknesses of the current wireless protocols. A threat model will be built to determine the security requirements of wireless networks against DoS attacks.

The solutions will be discussed by evaluating the IEEE 802.11 family of security protocols according to the classification framework and security requirements of the wireless networks.

C. THESIS ORGANIZATION

The rest of this thesis is organized as follows. Chapter II contains an overview and background of the Denial of Service attacks in both wired and wireless networks. Chapter III explains the step-by-step implementation of open-source test-bed environment and entities for the 802.1X protocol. Chapter IV provides a classification framework for all types of DoS attacks along with a detailed example of each attack type. Chapter V demonstrates various types of DoS attack applications by modifying the hostap source code. In addition, this chapter provides a threat model to determine the security requirements of WLANs for DoS attacks. Finally, Chapter V proposes solutions

by evaluating the 802.1x and 802.11 protocols. Chapter VI presents conclusions drawn from the experimentation and suggests areas for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND OVERVIEW OF THE DENIAL OF SERVICE ATTACKS

A. INTRODUCTION

This chapter will provide a brief overview of Denial of Service attacks in both wired and wireless networks. Since most of the DoS research has been done for wired networks, this chapter will focus primarily on wireless networks. Wireless security protocols, media access control, and authentication mechanisms will be examined to provide the background for further discussions of DoS attacks in WLANs.

B. THE HISTORY OF DENIAL OF SERVICE ATTACKS

A Denial of Service attack is a malicious behavior that prevents legitimate use of a network or computer. In so doing, it prevents authorized access to the services and delays time-critical operations. A DoS attack can be created by intentionally misusing the legitimate protocols, by wasting the restricted resources on servers or by exploiting flaws in the protocols. The most well-known and widespread method for mounting a DoS attack is the exhaustion of restricted resources, such as memory and bandwidth. DoS attacks can slow the network or completely destroy the connection between the users and resources.

The first documented DoS attack occured on November 3rd, 1988. Robert Morris Jr. created a worm and released it on the Internet. Many of the computers across the United States were affected [5]. They could not perform normal operations. Since the first computers had restricted amount of resources, the DoS attacks were based on resource destruction or resource allocation on a single computer.

This DoS attack has led the hardware and software companies to modify their systems. Companies subsequently released virus protection software and Intrusion Detection System (IDS) against attacks using malformed and malicious packets. Various types of firewalls have been developed to control ingoing and outgoing packets in an effort to protect against unauthorized access. However, while keeping pace with the development of the security tools, the DoS attacks became more sophisticated. The

attackers used more than one attacker computer to launch a combination of DoS attacks, known as Distributed Denial of Service (DDoS) attacks. In so doing, the attacker uses the combined power of many hosts to exhaust the resources of a victim.

The first DDoS attacks occurred in February 2000. They attacked the web sites of several large companies such as Yahoo.com, Amazon.com and CNN.com. The legitimate users were not able to access these web sites for several hours. The monetary damage caused by the attacks was approximately $1 billion [5].

Recently, the convenience of Wireless Local Area Networks (WLANs) has led to a proliferation of WLAN technology in the network market specifically in industrial and military sectors. Efficient use of free spectrum and cheap hardware are other reasons that WLANs are popular. One can set up a WLAN with an open-source implementation using only two wireless network cards. Since 802.11-based networks are used widely in the home, government and military, they are also attractive targets of various attacks, including DoS attacks.

The IEEE 802.11 wireless standard specifies both an authentication service and an encryption protocol, but researches have demonstrated how severely flawed these are. The attackers exploited these vulnerabilities to launch several types of attacks including a DoS attack. The IEEE 802.11i workgroup proposed the use of the IEEE 802.1X Port-Based Network Access Control Standard to meet the security requirements of the WLANs until they finish the 802.11i security protocol. However, Arunesh Mishra and William Arbaugh from the University of Maryland claimed that they could mount successful Denial of Service (DoS) attacks against 802.11X using commercially available client/supplicant with little trouble or development effort because of several design flaws of the 802.1X standard and the Extensible Authentication Protocol (EAP). While the 802.11i workgroup has released a draft to overcome these vulnerabilities for the WLANs. However, the draft’s implementation is not yet commercially available and nor tested on the market .

C. OVERVIEW OF THE MEDIA ACCESS CONTROL

The stations in wireless networks must agree on their access and their use of the shared media, which is radio frequency. This process is handled by the Media Access Control (MAC) protocol. The MAC protocol in wireless networks is a carrier-sense multiple access protocol with collision avoidance (CMAC/AD) techniques. Before transmitting, each station monitors cahnnel's radio frequency to determine whether or not another station is transmitting. If the channel is idle for an amount of time greater than the Distributed Inter-frame Space (DIFS), then the station is allowed to transmit.

When a receiving station receives the frame, it waits a specified amount of time, known as the Short Inter-frame Spacing (SIFS). Finally, the receiver will send an acknowledgment frame to the sender to indicate the successful reception of the frame. Figure 1 shows the exchange of the data and acknowledgment with the timing intervals.

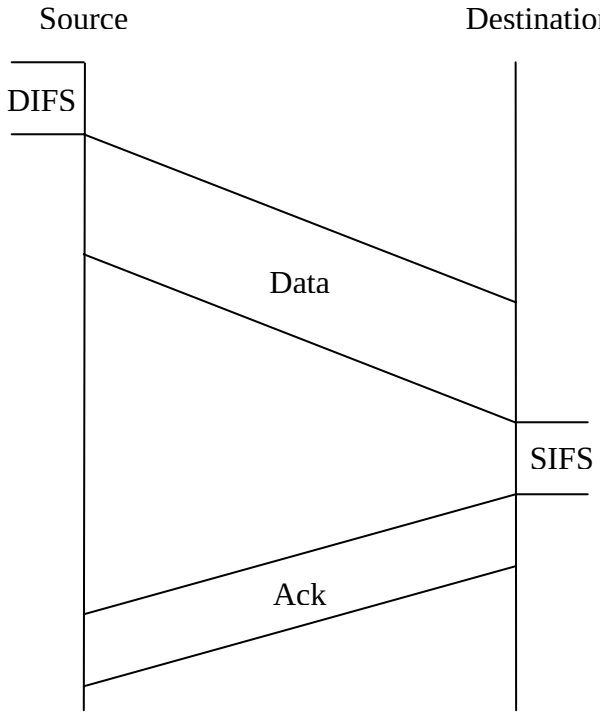


Figure 1. Data and Acknowledgment in IEEE 802.11

If the sender senses the channel is busy, then it will defer the access until the channel is clear. Once the channel is sensed idle for a time equal to the DIFS, then the sender waits a random back-off time as the channel is sensed idle. Finally, the station will transmit its frame.

The IEEE 802.11 MAC protocol uses a Request-to-Send (RTS) and Clear-to-Send (CTS) control frame to avoid collision and to reserve access to the channel. The sender will send an RTS frame to the receiver, before sending a data frame indicating the duration of the data and the ACK packet. The receiver will return the CTS frame

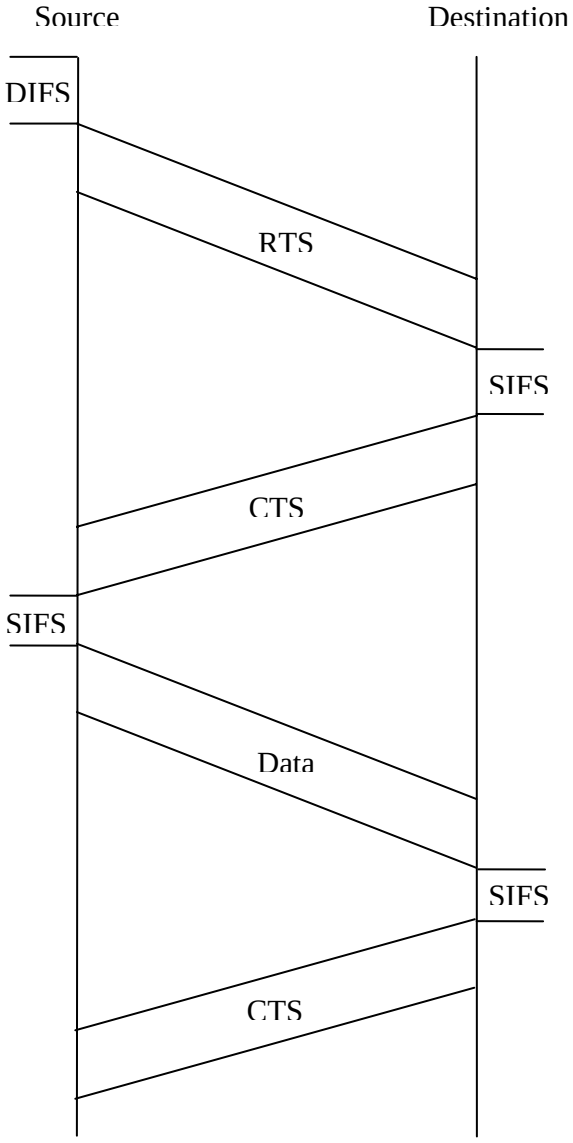


Figure 2. Collision Avoidance Using the RTS and CTS Frames

providing the transmission permission to the sender. In order to avoid interference, the other stations that receive the RTS and CTS control frame will not transmit. Figure 2 shows the exchange of these frames.

The RTS and CTS frames are used to avoid collisions in the IEEE 802.11 protocol. However, The DoS attacks can be achieved by intentional misuse of these control frames. In addition to these frames, three other control frames exist, including Power Save Poll (PS-Poll), Contention Free End (CF-End) and Contention Free Acknowledgment (CF-Ack) frames. Chapter IV will explain Denial of Service attacks based on the control frames, such as the RTS, CTS and PS-Poll frames.

D. OVERVIEW OF THE 802.11 AUTHENTICATION METHODS

Most of the known Denial of Service attacks in wireless networks are based on the functioning of the authentication and media-access control mechanisms. The control and management frames are heavily used for the authentication process between a wireless client and an access point. Sections D and E provide an overview of the authentication mechanisms defined in both the 802.11 and 802.1X standards. The overview of these protocols and frames will provide a better understanding of DoS attacks in wireless networks.

1. Open-System Authentication

The open-system authentication is known as a null-authentication process in which the station, or client, always successfully authenticates with the access point. The access point allows all users to authenticate successfully and use the network resources. Figure 3 illustrates the authentication sequence.

The access points and the users are configured to use the same Service Set Identifier (SSID) for successful authentication. The user will send a MAC authentication frame to the access point. The access point will response to the end user with the successful authentication frame indicating the end user is accepted. [2]

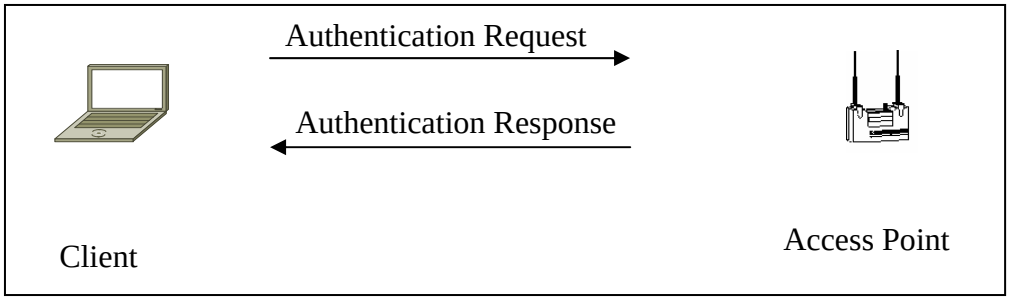


Figure 3. Open-System Authentication Schema

2. Shared Key Authentication

Shared-key authentication requires a shared key and a cryptographic technique for authentication between the client and the access point. This authentication method is based on a challenge-response message sequence in order to determine that the client knows the shared secret. Figure 4 illustrates the message sequence between the client and the access point. At first, the client will send an authentication request to the access point. A randomly generated challenge will be sent to the client by the access point. The client uses the shared secret to encrypt the random challenge and returns it to the AP. The AP will decrypt the result and compare it with the random challenge sent by the AP. If they are the same, then the access point will send a successful authentication message and will allow the client to use the network resources. [1]

The RC4 stream cipher is used for the cryptographic computation. The shared-key authentication method does not provide mutual authentication. Only the access point authenticates the client. So the client does not know whether or not it is communicating with the legitimate access point. This is one of the critical flaws in this authentication method. Rogue access points can be set up easily to launch denial of service attacks against the shared-key authentication method.

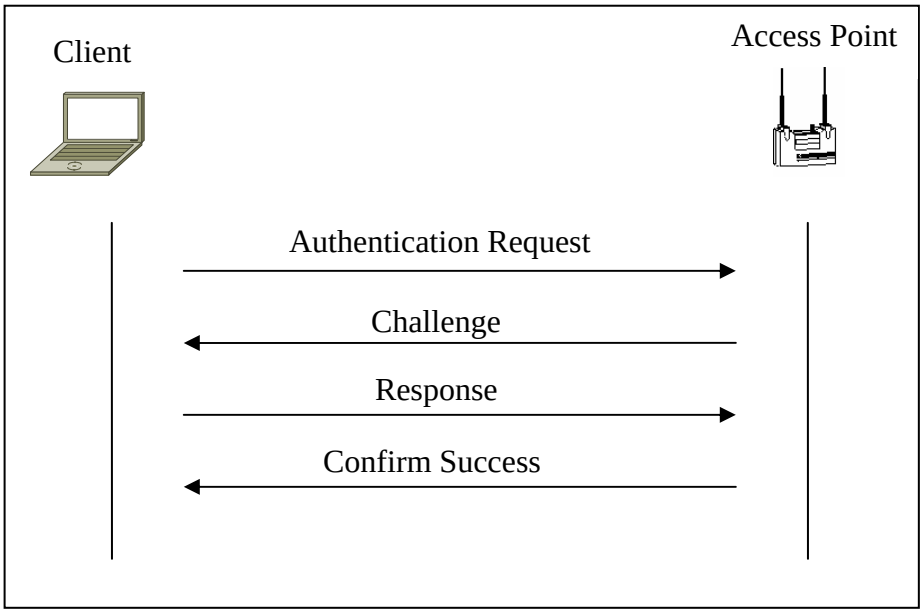


Figure 4. Shared-Key Authentication Message Sequence

E. OVERVIEW OF THE 802.1X AUTHENTICATION STANDARD

The concept of IEEE 802.1X is to provide a standardized security authentication method for IEEE 802 based WLANs. Three different entities are involved in the 802.1X authentication method.

- A *Supplicant:* An entity (client) at one end of a point-to-point LAN segment that is being authenticated by an authenticator (authentication proxy) attached to the other end of that link.[3]
- An *Authenticator:* An entity (Access Point) at one end of a point-to-point WLAN segment that facilitates authentication of the entity attached to the other end of that link (wireless client).[3]
- An *Authentication Server*: An entity that provides authentication service to an authenticator (Radius Server-see below).[3]

The Controlled/Uncontrolled Port model is used between the supplicant and the authenticator. The supplicant can use the uncontrolled port during the pre-authentication phase for sending the EAP frames. After a successful authentication, the supplicant is

authorized to communicate via the controlled port for access to network services. Three different protocols are used by the IEEE 802.1X.

- *EAPOL:* The Extensible Authentication Protocol over LAN (EAPOL) protocol carries EAP packets between the authenticator and the supplicant. So these entities can initiate the EAP authentication session by using the EAPOL protocol.
- *EAP:* The EAP protocol was originally developed for use with the point-to-point protocol (PPP) in RFC 2284 and has since been widely deployed with wireless networks. The IEEE 802.1X standard integrates the EAP protocol to provide an authentication mechanism for the IEEE 802.11 standard.
- *RADIUS*: The Remote Authentication Dial in User Service (RADIUS) is adopted between the authentication server and the authenticator. The EAP packets are encapsulated in RADIUS packets. A shared secret provides authenticity and integrity for the carried packets by using the HMAC MD5 hash algorithm.

After a brief explanation of the components and protocols used in 802.1X, the authentication message flow will be examined since most of the DoS attacks are based on flaws in the 802.1X authentication protocol. Figure 5 shows the message sequence in detail [4].

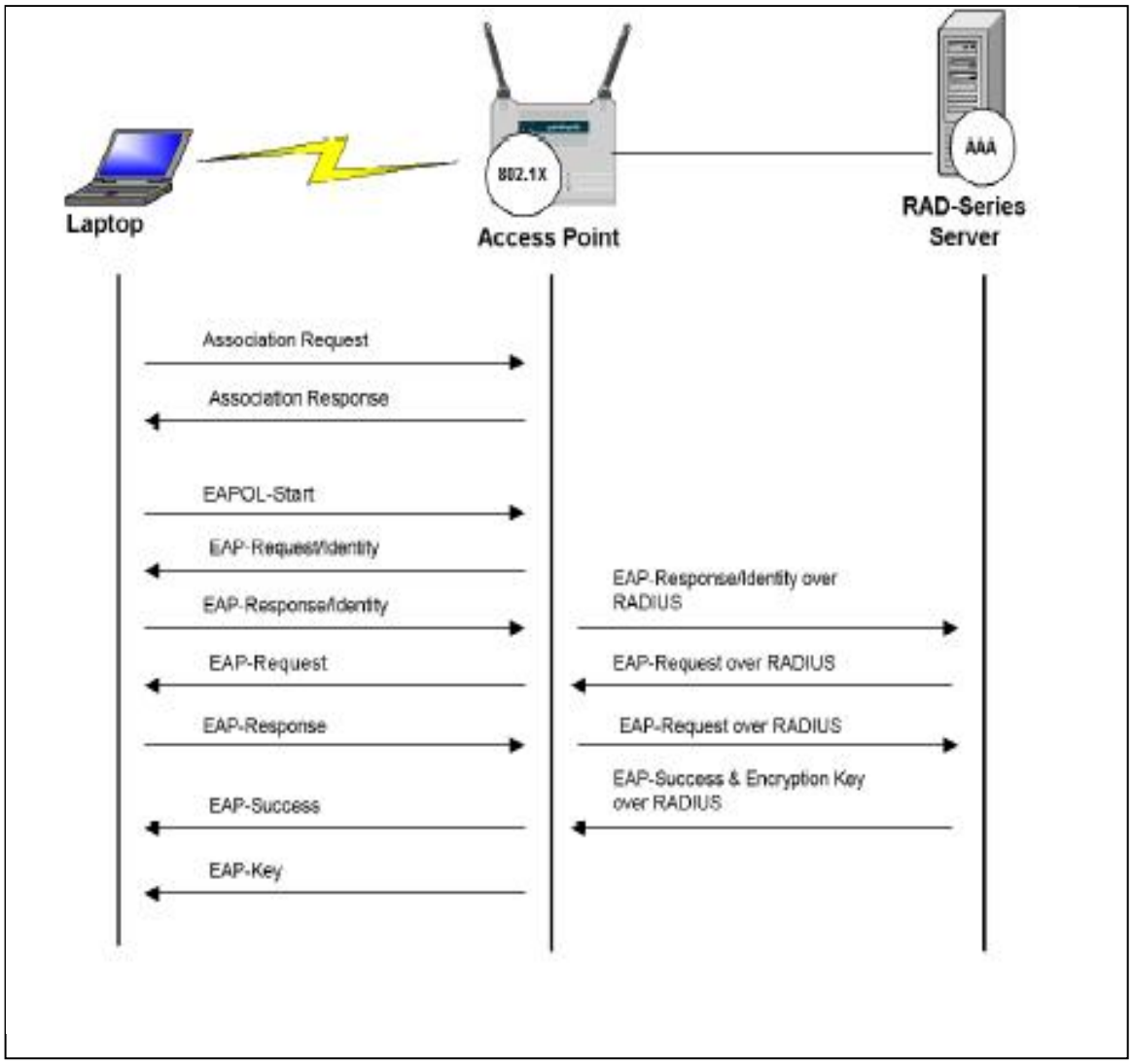


Figure 5. 802.1X Authentication Message Sequence (From Ref. 4)

At first, the supplicant will send an EAPOL-Start packet to initiate the authentication session.

The authenticator replies with an EAP-Request/Identity packet encapsulated in an EAPOL packet format.

The supplicant sends its identity credentials with an EAP-Response/Identity message to the authenticator. The authenticator receives this packet and encapsulates it

into the RADIUS Access Request-EAP/Identity Message. The authenticator will relay this RADIUS frame to the authentication server.

After the authentication server receives the RADIUS Access-Request Message, it confirms that the supplicant ID is valid.

The authentication server will send a challenge message based on the supplicant ID to request the supplicant certificate. The authenticator will receive this RADIUS-Access Challenge message and relay it to the supplicant.

The supplicant will respond with an EAP-Response message containing its credentials. If the authentication server validates the certificates then it will respond with an EAP-Success message. After successful authentication, the supplicant can use the controlled port and the network resources.

Since the management frames used in 802.1X authentication session do not enforce integrity and privacy, the protocol is vulnerable to the DoS attack. Chapter IV will discuss how to circumvent these frames.

III. AN OPEN-SOURCE WIRELESS PROTOCOL TEST-BED

A. INTRODUCTION

The IEEE 802.1X standard addresses some of the IEEE 802.11 security vulnerabilities. However, the 802.1X security standard is still vulnerable to Denial of Service attacks. This thesis primarily analyzes various types of DoS attacks based on data, control, and management frames.

For verification and analysis of DoS attacks against WLANs, an 802.1X test-bed was built on an IEEE 802.1b wireless LAN. This chapter explains how to build and configure the 802.1X entities: the supplicant, the authenticator and the authentication server. The open-source software was used on the Linux Operating System environment for availability and ease of source-code manipulation.

B. 802.1X AUTHENTICATION TEST-BED

This thesis is based on open-source software because it is easier to demonstrate DoS attacks and to analyze the results of this experiment. This section explains how to combine the Linux environment with the open-source software as illustrated in Figure 6.

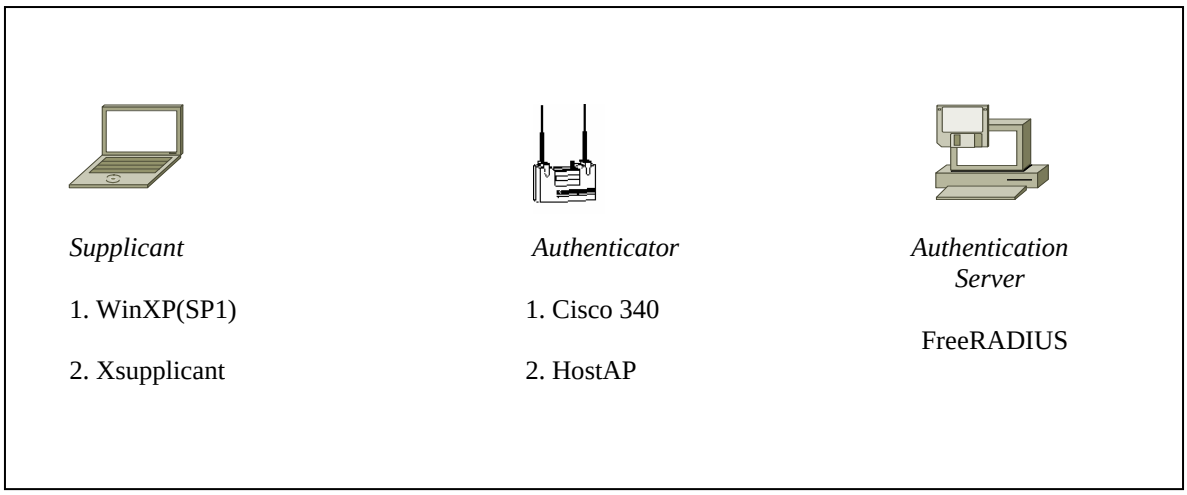


Figure 6. 802.1X Test-bed Schema (From Ref. 10)

1. Elements of the Authentication Test-Bed

The security framework of the 802.1X standard [14] consists of three main entities: the supplicant, authenticator and authentication server.

1). A *Supplicant:* An entity that desires to use the services offered by the authenticator. The client side of the wireless network is named as the supplicant. It is an entity at the one end of the network that is authenticated by the authentication server on the other end of the network.

2). An *Authenticator:* An entity at one end of a point-to-point LAN segment that facilitates authentication of the entity attached to the other end of that link.[3] The authenticator has two roles: one before the authentication and the other after the authentication. The authenticator relays the authentication packets between the Supplicant and the Authentication server. After a successful authentication takes place, the authenticator provides network connectivity to the supplicant independent of the authentication server.

3). An *Authentication Server*: An entity that provides authentication service to an authenticator. This service determines from the credentials that the supplicant provides, whether the supplicant is authorized to access the network services provided by the authenticator [3]. An authentication server is the authority in a network that decides the access of the supplicant according to its credentials. This server is only needed for the authentication. After a successful authentication, the server is dormant until another supplicant wants to use the network.

2. Authentication Methods

While building the test-bed, the most secure authentication method must be selected. One of the main factors in choosing the right authentication method for the EAP protocol is the ability to provide mutual authentication between the 802.1X entities. This is true because mounting attacks against WLANs is more difficult if the security protocol provides robust mutual authentication.

Two types of authentication methods are used and supported by the current authentication servers. The first one is Cisco's Lightweight Extensible Authentication Protocol (LEAP); the second one is the Transport Layer Security (TLS). The EAP-TLS authentication method was chosen for mutual authentication. TLS requires a public-key infrastructure (PKI) for certificate-based authentication.

3. Hardware and Software Configuration of the Test-Bed

There are three entities in the 802.1X authentication mechanism: the supplicant (mobile client), the authenticator (access point) and the authentication server (free radius).

a. Supplicant

Supplicant is an entity that requests network access and is being authenticated by an authenticator. A Pentium III laptop with PCMCIA support is used as the supplicant for an Open-1X test-bed. A D-Link DWL-650 wireless network interface card is used for wireless communication between the supplicant and the authenticator.

Two operating systems were considered to host the supplicant: Windows XP and Linux Red Hat. Both of them have advantages and disadvantages. The supplicant in Linux Red Hat provides a wide range of tools for the supplicant's configuration; in addition the supplicant in Windows XP is easy to implement. Since this thesis focuses on the Denial of Service attack, which does not require any configuration of the supplicant, Windows XP is used for an Open-1X test-bed. In this section, both the Windows XP client and the xsupplicant will be explained in detail.

(1) Windows XP SP1: The Microsoft Windows XP with a Service Pack 1 (SP-1) operating system is used for its embedded IEEE 802.1X supplicant support. A D-Link DWL-650 network interface card is used for wireless communication. The driver can be easily downloaded and installed from http://support.dlink.com/. After installation of Windows XP SP1, the public key certificate and private key of the client are created. The public key certificate (PKC) of the root certificate authority (Root-CA) is imported and installed. Appendix B provides the details on installing the certificates and configuring of the 802.1X protocol for the supplicant.

(2) Xsupplicant: The Linux Red Hat operating system can host the Xsupplicant on a mobile laptop. The same D-Link DWL-650 NIC is used as a wireless interface card. DWL-650 does not officially support Linux drivers. However, the chipset (Intersil Prism2) is supported by Linux Red Hat 8.0 via ornico_cs driver.

The three dependent libraries should be built and installed before the Xsupplicant source code:

- OpenSSL 0.9.7 (http://www.openssl.org/)
- Libpcap 0.7.1 (http://www.tcpdump.org/)
- Libnet 1.1.0 (http://www.packetfactory.net/libnet/)

All these components should be downloaded and uncompressed into the */usr/src/xsup* directory for the installation. The "readme" and "install" documents furnish adequate information in order to build the libraries. The following commands are sufficient for building and installing the required libraries for the Xsupplicant.

cd /usr/src/xsup/<directory name>
. /configure
make
make install

The Xsupplicant source code can be downloaded from the sourceforge.net website (http://sourceforge.net/projects/open1x/). The compressed source code (tarball) should be uncompressed into the */usr/src/xsup* directory. The following commands suffice for the default installation of the xsupplicant source code.

cd /usr/src/xsup/xsupplicant
. /configure –enable-full-debug
make
make install

The “enable-full-debug” flag causes critical information and run-time errors to be printed during configuration. It is crucial to use this flag in order to determine whether or not the dependent libraries are found by the xsupplicant source code.

After installing the dependent libraries and xsupplicant source code, the certificates created by Certificate Generator should be copied into the designated directory, as defined in the configuration file (*1x.conf*). This configuration file must be copied into the */etc/1x/* directory since the xsupplicant daemon requires the *1x.conf* file to be in that directory by default. Finally, the xsupplicant daemon can be activated with the following commands.

iwconfig wlan0 essid test

xsupplicant –I wlan0

b. Authenticator

The authenticator is an entity at one end of a LAN segment that facilitates authentication of the entity attached to the other end of that LAN. In this context, an authenticator forwards 802.1X frames of a supplicant to an authentication server for authentication. The authenticator provides network connectivity to the supplicant via a controlled port after a successful authentication. In order to achieve this, a dual-port model is used. Figure 7 shows the dual-port concept employed in IEEE 802.1X. The authenticator has two ports for external access to the network that it is protecting: The Uncontrolled Port and the Controlled Port. The Uncontrolled Port filters all traffic and allows only the EAP packets to pass for the authentication. After successful authentication, the supplicants can use the Controlled Port to send regular network traffic through the authenticator.

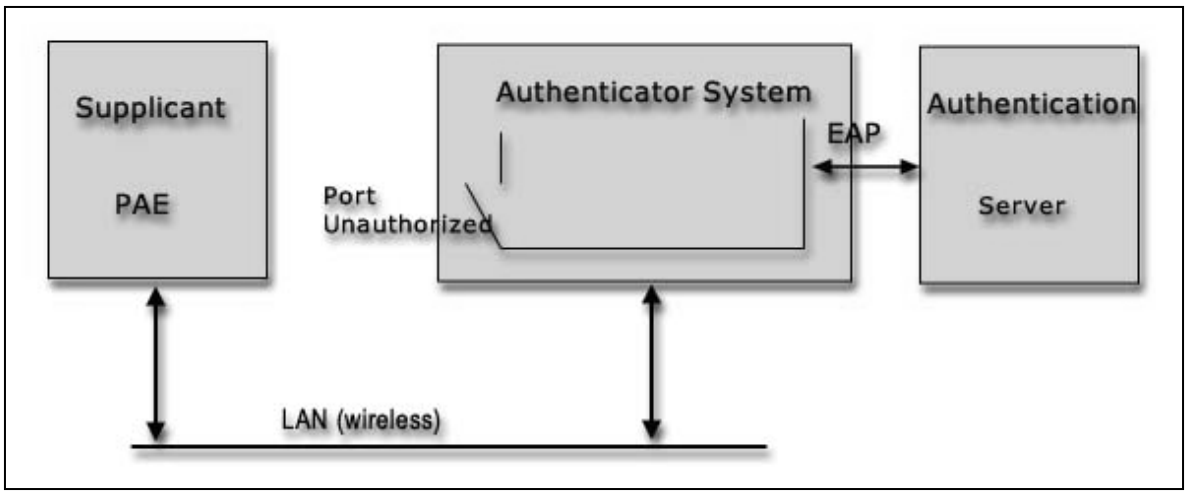


Figure 7. 802.1X Authenticator Dual-Port Concept (From Ref. 10)

For this thesis, the authenticator is the most important entity in the open-source 802.1X test-bed, since it enables 802.11b access point functionality using the firmware of the Intersil chipsets for time sensitive tasks. All other functionality is handled by the Hostap driver, including WEP and passing frames off to an authentication server.

(1) HostAP: The Host AP driver is a Linux driver for wireless LAN cards based on Intersil's Prism 2/2.5/3 chipset. Since D-Link DWL 650 wireless NIC is based on the Prism 2 chipset, the Hostap driver will support it. The driver supports Host AP mode and does not require any special firmware for the D-Link DWL 650 wireless NIC. It performs IEEE 802.11 management functions and acts as an access point. In addition, it implements the following IEEE 802.11 functions:

- Association
- Authentication
- Data transmission between two wireless stations
- Power saving mode signaling
- Frame buffering.

An IBM Think-Pad 600 Pentium II laptop and a D-Link DWL 650 wireless NIC are the hardware components of the authenticator. Linux Red Hat 9.0 with kernel 2.4.22 is installed to support the Hostap driver.

The Wireless Extensions v15 and v16 patches are not necessary for the Linux kernel version 2.4.22. Two kernel-configuration options must be enabled during the configuration of the kernel (2.4.22): the wireless LAN (non-hamradio) option for Hostap support (Figure 8) and the 802.1d Ethernet Bridging option (Figure 9) for bridging support between the wireless and wired interface. These graphical configuration menus will be displayed after the *xconfig* command of the kernel configuration process. These options are not enabled by default. The authenticator would not function properly without these support options.

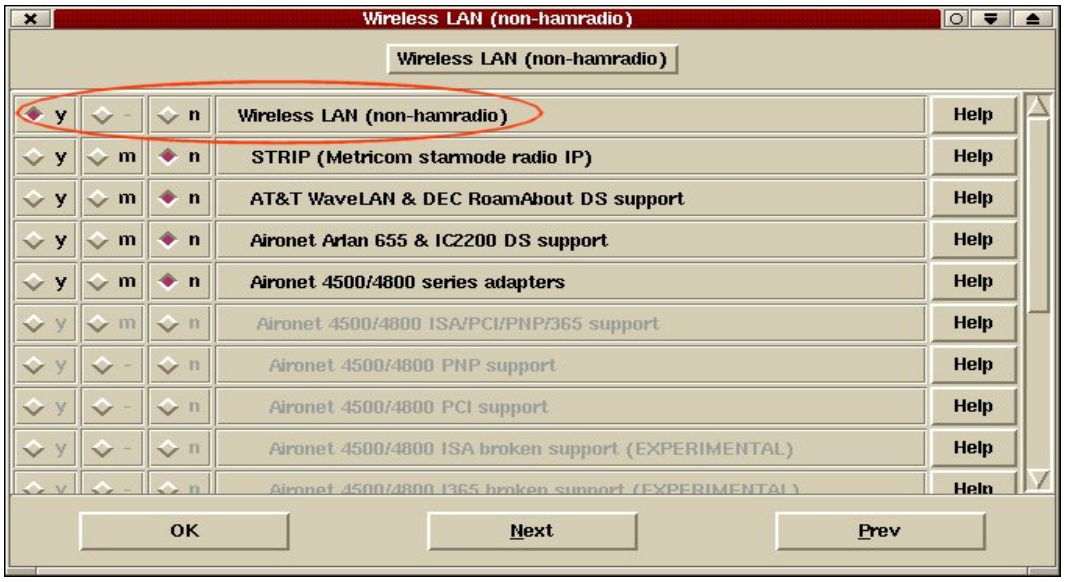


Figure 8. Wireless LAN (non-hamradio) Option

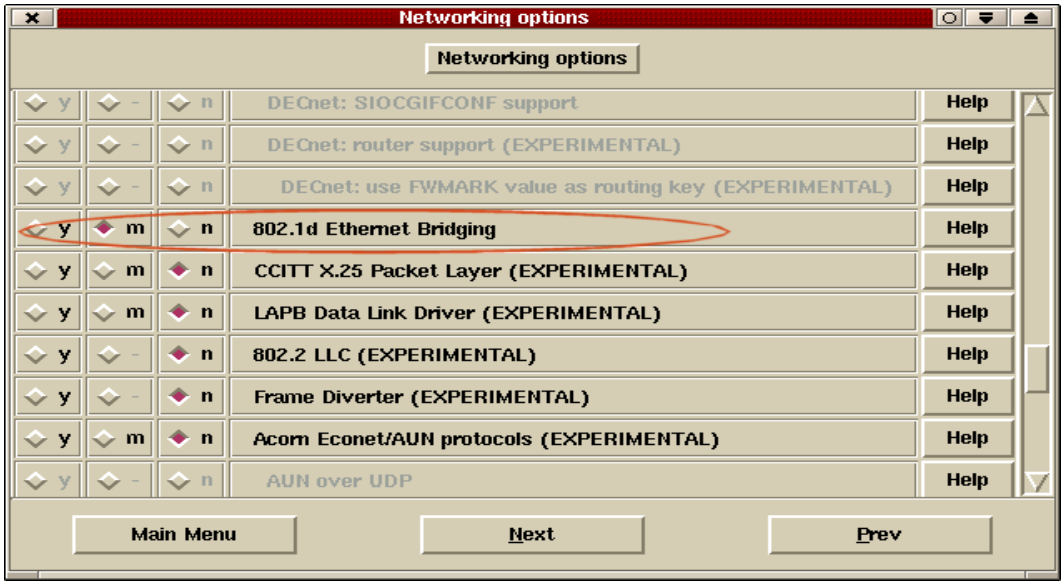


Figure 9. 802.1d Kernel Bridging Support

Before version v0.1.0, the Hostap driver used to be distributed as one tarball. Now the software is separated into three components. Since the Hostap v0.1.1 driver is used for the test-bed, the following three Hostap components and Wireless Extension tools were installed on the authenticator:

- Hostap-driver 0.1.1
- Hostapd
- Hostap-utils
- Wireless Extension Tools v25

The Wireless Extensions Tools v25 (http://pcmcia-cs.sourceforge.net/ftp/contrib/) were downloaded and uncompressed into the /usr/src/ directory tree. The source code was installed by the command sequence below.

. /configure

make

make install

The Hostap driver, utilities and daemon source code (http://hostap.epitest.fi) were downloaded and uncompressed into the /usr/src/ directory tree. The Hostap driver was configured by the commands below.

tar –zxvf hostap-driver-0.1.1.tar.gz

. /configure

After the configuration of the driver, a *Makefile* was created for this specific configuration. The KERNEL_PATH variable was set to the kernel configured for the access point in the */usr/src/hostap/Makefile* file.

 # Edit this path to match the system (It should point to the root directory

of the Linux kernel source.)

KERNEL_PATH=/usr/src/linux-2.4.22

 # Leave this blank for kernel-tree PCMCIA compilations

PCMCIA_PATH=

Since the Hostap requires kernel support, the Hostap source code must be built and installed after the proper configuration and required modification to the *Makefile*. The “Extra flag” option was required in order to support the 802.1X functionality for hostapd daemon.

make pccard EXTRA_CFLAGS=”-DPRISM_HOSTAPD”

make install_pccard

The Hostap utility and daemon components were built and installed to support the 802.1X authenticator functionality with the following sequence of commands:

configure

make

make install

The authenticator supports bridging since the 802.1d Ethernet Bridging option was checked during the kernel configuration. Bridging will provide

communication between the wireless and wired segments of the network. Otherwise, the authenticator cannot relay the regular network packets between the supplicant and the network. There must be two interfaces in the authenticator: an Ethernet interface (eth0) connecting the wireless segment to the wired network and a Wireless interface (wlan0) acting as an access points. The following series of commands are used to establish the bridging between the two interfaces.

ifconfig wlan0 0.0.0.0

ifconfig eth0 0.0.0.0

brctl addbr br0

brctl addif br0 eth0

brctl addif br0 wlan0

ifconfig br0 XXX.XXX.XXX.XXX up

Both interfaces' IP addresses must be set to zero and assigned to the bridge interface as defined above. The bridge interface (br0) is a logical interface rather than a physical one like eth0 or wlan0. The AP bridges packets between the Ethernet and wireless LANs and can be reached with the IP address XXX.XXX.XXX.XXX from either network. When the AP reboots, the bridging between the interfaces should also be reestablished.

After installing all the components and establishing the bridging, the authenticator was ready to run the *hostapd* daemon to serve as an 802.1X compliant access point. The *hostapd* daemon accesses a configuration file known as *hostapd.conf* to retrieve the parameters of the wireless network. Appendix C provides an example of this configuration file. The *hostapd* daemon is launched with the following command sequence.

cd /usr/src/hostap/hostapd

. /hostapd –d hostapd.conf

(2) D-Link DWL 7000AP: Since this Access point supports the 802.1X authentication protocol, it is chosen to be used as the authenticator of the test-

bed. The configuration of the access point was simple since it has a web-based configuration tool. Appendix C explains and illustrates the configuration of DWL-7000AP.

c. Authentication Server

Since FreeRADIUS (http://www.freeradius.org/) is the only available open-source authentication server tool that supports Linux, it was used by the test-bed for this thesis. FreeRADIUS supports the EAP-TLS authentication method as an embedded module.

One of the latest versions of the Red Hat Linux (Red Hat 8.0) was used as the operating system for the authentication server. The newer versions of FreeRADIUS are compatible with Linux Red Hat 8.0. In order to run the FreeRADIUS server on Linux Red Hat 8.0, the following software should be downloaded and installed.

- OPENSSL 0.9.7
- OPENSSL SNAP-20020227
- OPENSSL 0.9.7-beta3
- FREERADIUS-0.9.2

Three different versions of the OpenSSL are needed throughout the whole process. The stable version of OpenSSL (OPENSSL 0.9.7) is required to build most of the FreeRADIUS software. A recent snapshot version of OpenSSL (OPENSSL SNAP-20020227) is required to build the EAP/TLS modules. OpenSSL 0.9.7-beta3 is the third version of the OpenSSL, which is used to create the certificates.

The other open-source software package that must be installed is FreeRADIUS-0.9.2. After successfully installing all of the software, the Linux computer becomes the authentication server of the test-bed. The installation procedures of the necessary software come with the downloadable tarball. The following paragraphs emphasize the important details of the installation process.

(1) OPENSSL 0.9.7: OpenSSL 0.9.7 (http://www.openssl.org) is used for building FreeRADIUS. After downloading and uncompressing the tarball, the following sequence of commands is used to build the software.

cd openssl-0.9.7

. /config

make

make test

make install

These commands build and install the OpenSSL-0.9.7 in the default location, which is */usr/local/ssl* for Linux Red Hat 8.0. For the test-bed, the default location was used.

(2) OPENSSL SNAP-20020227: The snapshot version of the OpenSSL is used to load the EAP-TLS module in the FreeRADIUS source code. After downloading and uncompressing the OPENSSL SNAP-20020227 tarball, the source code was built and installed by using the commands below:

. /config –prefix=/usr/local/openssl shared

make

make test

make install

One of the most important parts of this installation is paying attention where the software is installed. If the config command was used without any switches, this version of OpenSSL would be installed to the default location and overwrite the stable version previously installed.

The final part of the installation is checking and verifying that *libssl.so* and *libssl.so.0* are symbolically linked to *libssl.so.0.9.8* while *libcrypto.so* and *libcrypto.so.0* are symbolically linked to *librypto.so.0.9.8*. These files can be found under the *lib* directory of the snapshot version of OpenSSL.

(3) OPENSSL 0.9.7-beta3: This version of the OpenSSL is necessary to generate the certificates used by the wireless network. This software can be installed on another computer and the certificates can be generated in a more secure location in real-life applications. However, to keep the process simple, the authentication server computer was used as the certificate generator as well. After downloading and uncompressing the software, the following commands were used to install this version of OpenSSL:

. /config –prefix=/usr/local/openssl-certgen shared

make

make test

make install

The final step of the installation is checking and verifying the same sym links of the specific lib files, which were mentioned in the installation of the SNAP version. These files can be found under the *lib* directory of the beta version of OpenSSL. Appendix A provides the OpenSSL configuration file for the certificate generator.

(4) FREERADIUS-0.9.2: The latest version of FreeRADIUS (FreeRADIUS 0.9.2) modules were downloaded and uncompressed into the */root/downloads* directory. The FreeRADIUS source code was configured before building by using the commands below:

cd /root/downloads/freeradius-0.9.2

. /configure –sysconfdir=/etc

The configuration script prepares the *makefile* to build the source code. The OpenSSL version that will be used to create the EAP-TLS modules is different than the default OpenSSL, the EAP-TLS *makefile,* which was placed in */root/downloads/freeradius-0.9.2/src/modules/rlm_eap/types/rlm_eap_tls/* directory, was modified as defined in Appendix D. After modifying the *makefile,* the FreeRADIUS can be installed by using the following commands:

make

make install

After building and installing the FreeRADIUS software, the configuration files of the radius server should be modified to enable the 802.1X authentication scheme. There are three configuration files that should be modified: *radiusd.conf, users,* and *clients.conf.* The *radiusd.conf* file is modified to make EAP-TLS work properly. The *users* file is modified to include pointers to client certificates. The c*lients.conf* is modified to allow access to the access points of the wireless network to request authentication. A test user may also be temporarily added to the *users* file to check the functionality of the authentication server without using the other components. Appendix D provides a copy of all three configuration files.

For the session-key management, two random files must be created. These files only need to contain random characters. For this particular occasion, the *date* command was used to create these two random files.

date > /etc/1x/random

date > /etc/1x/DH

If everything has been installed and configured correctly, the FreeRADIUS should be ready to run and to authenticate the legitimate users via the EAP-TLS. A wrapper script is created to run FreeRADIUS with the correct SSL libraries of the OpenSSL snapshot version. Appendix D provides a copy of this script (*run-radiusd*).

The FreeRADIUS may be run from the shell by typing *run-radiusd –X –A*. (This runs the script that is mentioned in the preceding paragraph). After seeing that the server is running correctly and listening for requests, the test user can be employed to test the server. If the result of the test is good and an "Access-Accept" message is returned, then the server is running well and the test user may be deleted.

4. EAP-TLS Authentication Method

This section discusses the Extensible Authentication Protocol Transport Layer Security (EAP-TLS) briefly since EAP-TLS protocol was examined in [1] in detail.

EAP-TLS authentication is based on the 802.1X/EAP architecture. The three entities involved in the 802.1X/EAP authentication process are supplicant (the end entity), the authenticator (the access point), and the authentication server (RADIUS server). The supplicant and the RADIUS server must support EAP-TLS authentication. The access point must support the 802.1X/EAP authentication process. For example, a Cisco Aironet access point that supports the EAP-TLS authentication protocol can be used in the 802.1X test-bed.

The 802.1X test-bed requires the EAP-TLS authentication protocol for mutual authentication and key exchange between the entities. EAP-TLS (RFC-2716) uses the TLS protocol (RFC-2246), which is the latest version of the Secure Socket Layer (SSL) protocol. TLS provides a mechanism for both user and server authentication and for dynamic session key generation in order to use certificates. Figure 10 illustrates the general schema for a message exchange between the entities.

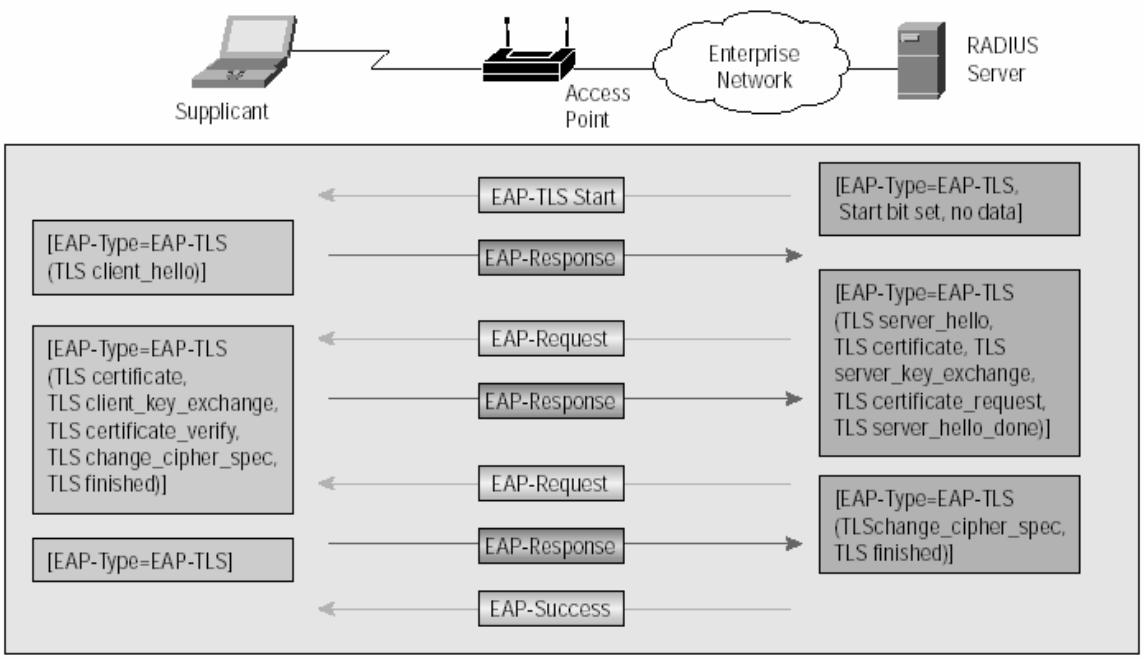


Figure 10. EAP-TLS Message Exchange (From Ref. 15)

5. X.509v3 Certificates

The EAP-TLS authentication method is certificate-based. This method uses X.509v3 certificates, which can be generated by using the OpenSSL software. Appendix A covers the OpenSSL configuration file of the certificate generator.

X.509v3 certificates contain the public key of the supplicant with some additional information. The certificates are created for the Root Certificate Authority, authentication server and the supplicant. The Root CA certificate is created first; the other certificates are digitally signed by the private key of the CA.

Three certificate generation scripts will be used to create all the necessary certificates. Appendix B provides a copy of all three certificate generation scripts with an XP specific extension file that is necessary to generate a certificate. One of the most important requirements when creating the certificates is to verify that the client name on the certificate matches the names in the *users* configuration file. Another critical point is to assign a password that is different from the default password (default is “whatever”) during the certificate generation process.

The first certificate to be generated is the root certificate. This certificate will be used to sign the other two certificates. So the generation sequence will be as follows:

./CA.root

./CA.srv <servername>

./CA.clt <clientname>

After all three scripts are run, the folder will contain 12 different certificates. The extensions of the certificates will be ".p12, .der, .pem". The two certificates that will be used by the FreeRADIUS are <s*ervername*>*.pem* and *root.pem*. The WinXP supplicant client requires <c*lientname*>*.p12* and *root.der*. So those two certificates should be copied to the supplicant and installed. The details of the certificate generation process can be found on the following web sites:

http://www.impossiblereflex.com/8021x/eap-tls-HOWTO.htm

http:// www.missl.cs.umd.edu/ wireless/eaptls/

6. Validation

After installing all the software to run the test-bed, the system was tested by using a packet capture software called Ethereal (www.ethereal.com). Ethereal captures all the network traffic. The user may select the network adapter to filter the traffic. Two laptops were used in the validation process. One of the laptops (Dell Insiron-5100) was configured to capture the wireless packets in promiscuous mode with Ethereal. The other laptop (HP Compaq pavilion Ze5400) was used as the supplicant.

The client certificates were installed on the supplicant, and the wireless card was enabled to initiate the authentication process. All the traffic concerning the authentication process was captured by Ethereal which was running on the other laptop. The packets captured by Ethereal, shown in Figure 11, verified a successful authentication process. Appendix E provides the successful logs of both the authentication server and the authenticator.

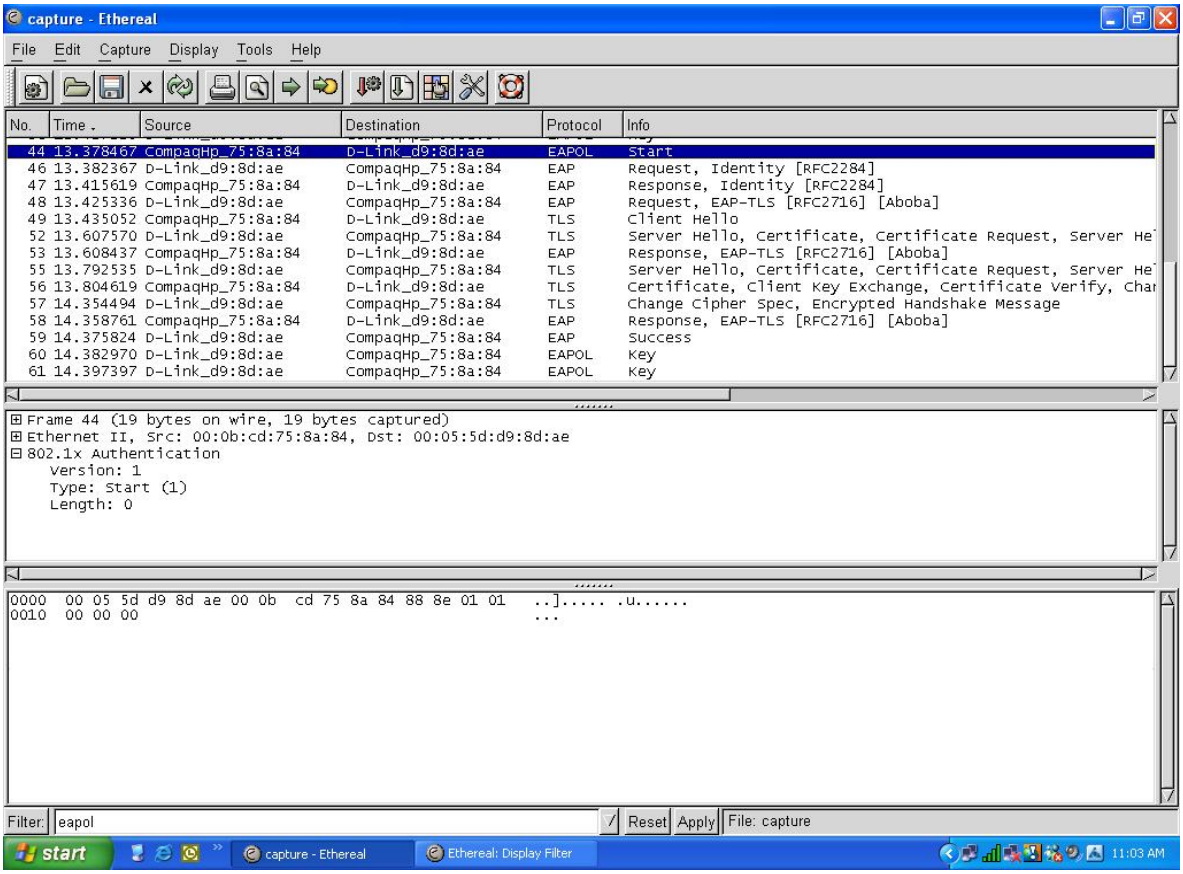


Figure 11. Captured Packets showing a successful authentication

THIS PAGE INTENTIONALLY LEFT BLANK

IV. A TAXONOMY OF THE DENIAL OF SERVICE ATTACKS

A. INTRODUCTION

DoS attacks can be classified in many different ways based on attack dynamics and the consequences. This chapter presents a taxonomy of DoS attacks based on the types of system flaws exploited by the attackers.

Typically, security threats are evaluated with respect to a particular protocol (application). Similarly, the DoS attacks can be classified into two main categories: attacks on the host operating system (OS) or network infrastructure and attacks that exploit specific weaknesses of the target protocol. The first category of attacks arise from implementation flaws of the network or the improper usage of the operating system and network components including bandwidth, buffers, and the TCP/IP stack. The second category of attacks exploits the vulnerabilities in the protocols to prevent the victim from using the network resources.

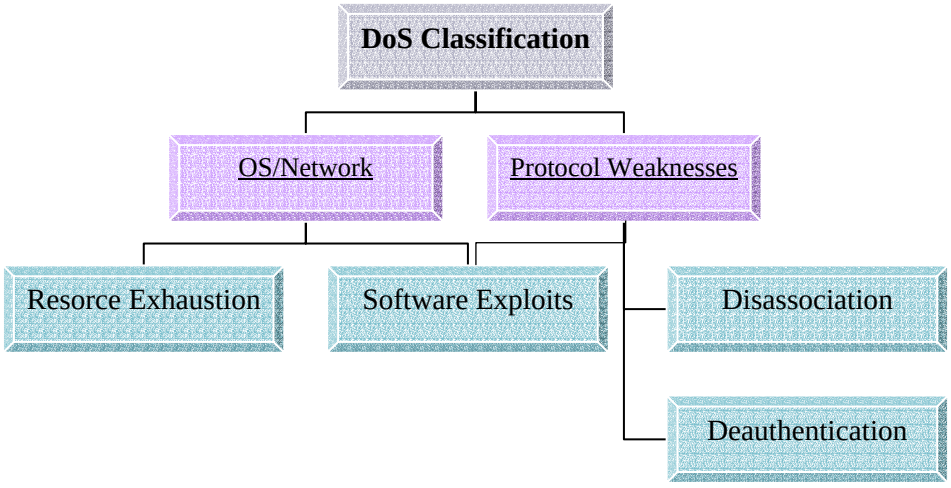


Figure 12. Taxonomy of DoS Attacks

These main categories are branched into the sub-categories to discuss the attacks based on the specific flaws in detail. OS/Network-based DoS attacks are divided into resource exhaustion and software exploits sub-categories. Protocol weakness includes deauthentication, disassociation, and software exploits sub-categories. Since the software

exploit DoS attacks can be part of the operating system or the protocol, both these main categories share the software exploit subcategory. The characteristics of each sub-category will be explained in following sections. Figure 12 illustrates the taxonomy resulting from the general classification framework.

Since it is difficult to find a solution to many different types of DoS attack, this classification framework will help us to generalize the solutions for each category. When a new DoS attack is encountered, the specification of this attack can easily be identified with this schema.

B. PROTOCOL WEAKNESESS DENIAL OF SERVICE ATTACKS

The protocols used in wireless networks such as 802.11 and 802.1X are severely flawed. Well-known vulnerabilities of the authentication phase and a lack of authenticity verification of the management and control frames allow for DoS attacks. The attackers take advantage of the improper design of the protocol or lack of security mechanisms to launch this type of attack. Protocol weakness DoS attacks are divided into two sub-categories: Disassociation and Deauthentication DoS attacks. The characteristics of each sub-category will be discussed in the following section.

1. Deauthentication DoS Attacks

The clients in wireless networks need to authenticate and associate themselves with the AP in order to receive the network connectivity. The frames used during the authentication process are not encrypted. The confidentiality and integrity of these management frames are questionable. During the authentication phase, the attacker can spoof these frames to launch the DoS attacks.

There are three different authentication mechanisms in wireless networks, but open-system and shared-key authentication mechanisms share the same characteristics in DoS attacks. So DoS attacks based on the authentication flaws can be divided into two groups: shared-key and 802.1X DoS attacks.

DoS Attacks in Shared-Key Mechanism: An 802.11 client must authenticate itself to the AP in order to initiate the communication between the client and network.

Either the client or the AP can request deauthentication from the other to end the communication.

An attacker takes advantage of this vulnerability to launch the deauthentication DoS attack. The attacker can easily spoof this message, pretending to be an access point or a client and send this message to the victim. The victim, either the client or the access point, will enter the deauthentication phase and stop the communication until the authentication is reestablished. The attacker can launch this attack repeatedly and prevent the victim from transmitting or receiving data. Figure 13 shows a Deauthentication attack message sequence.

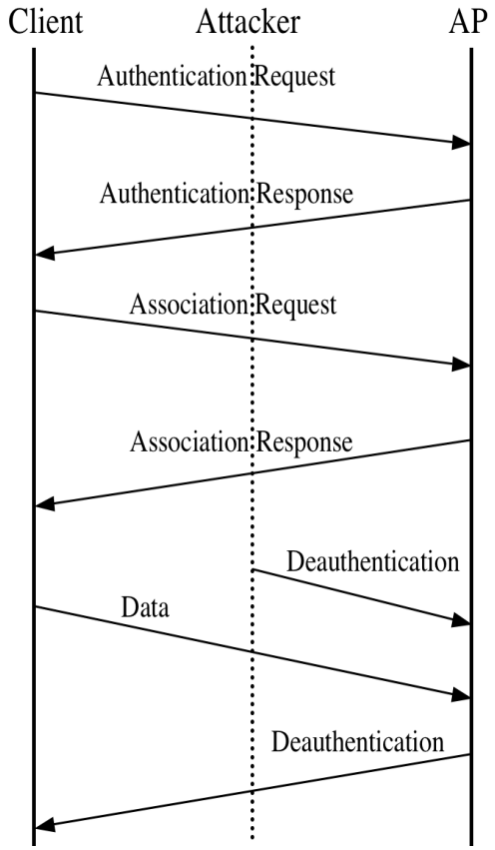


Figure 13. Deauthentication DoS Attacks in Shared-key Mechanism (From Ref. 9)

DoS Attacks in 802.1X Mechanism: The deauthentication DoS attacks launched in WLANs that use 802.1X authentication protocol are based on EAPOL Logoff or EAP Failure packets. The attacker spoofs the client's MAC address and sends the EAPOL

Logoff packet to the AP. Subsequently, the AP will terminate the connection with its authenticated client.

When a legitimate client is in the process of authentication, the attacker can send the EAP Failure packet to the AP spoofing the victim's MAC address. This packet will cause the AP to terminate the legitimate client's authentication process. The attacker must catch the victim during the authentication process. The attacker can use passive packet sniffer programs to capture the association response frame, which indicates the beginning of the authentication phase.

2. Disassociation DoS Attacks

This type of DoS attacks prevents any connection between the user and services from being established even though resources are available. The result is disruption of the communication. The victim system is denied from using the services until it reassociates with the services. This sub-category includes disassociation and power saving DoS attacks.

Power Saving DoS Attack: The 802.11 protocol has a power conservation functionality. The clients can enter the sleep mode after they send the message to the AP to activate the sleep mode. During this phase, the clients cannot receive or transmit data through the AP. The AP buffers all incoming traffic for the sleeping client. After the client returns to normal mode, it sends a polling message to the AP for the pending traffic. If there is any traffic in the buffer for the client, then the AP will deliver the data to the client and discard its buffer.

Since the polling message is not authenticated with any keying material, the attacker can easily spoof this message and cause the AP to discard the client's pending traffic while it is in sleep mode. On the other hand, the AP periodically broadcasts a packet known as a Traffic Indication Map (TIM). The TIM packet indicates the presence of the buffered data for the client. If the attacker spoofs this packet, then the client can be convinced that there is no pending data for itself. These are the two variations of the Power Saving DoS attacks in wireless networks. [9]

Disassociation DoS Attack: The attacker can exploit a similar vulnerability as with the deauthentication DoS attack in the association phase. A client can authenticate itself with many of the APs but it must associate with only one of the APs which is responsible for forwarding and receiving the client's packets. The 802.11 protocol specifies a disassociation message, which allows the client to switch between the available APs. An attacker can spoof the legitimate AP MAC address and send a disassociate message to the clients. The client then must reassociate with the AP to send and to receive the data. This attack method is less efficient than the deauthentication DoS attacks since it requires less work to return to the associated state.

C. OS/NETWORK DENIAL OF SERVICE ATTACKS

Operating system and network vulnerabilities make attacks of this category possible. Sometimes the attacker subverts the expected behavior of the system into well-designed attacks since the implementation of the OS or network is flawed. This category includes the resource exhaustion and implementation DoS attack sub-categories.

1. Resource Exhaustion DoS Attacks

A resource exhaustion attack directs a flood of packets to the victim aimed at overwhelming a certain system resource of the victim. Most "flooding attacks" mentioned in the literature belong to this category. The target system resources include network bandwidth, frequency spectrum, disk space, host memory and CPU availability. This category is a well-known DoS attack type against popular web sites since it does not require sophisticated methods.

The attacker takes advantage of the expected behavior of the protocols to bring down the victim's system. For example, a three-way handshake is the first step in requesting a web page. The attacker can send excessive packets that the victim cannot handle by consuming the limited connection queue. So the victim's server cannot respond to the legitimate client's requests. The difference between this category and the others is the rate at which the attacker what he does, not what he does.

Most of today's computers with high power CPU and memory can handle the resource exhaustion DoS attacks from one target. However, with the continued development of computer hardware and security tools, the DoS attacks became more

sophisticated. The attackers can use more than one computer to launch a combination of DoS attacks, known as Distributed Denial of Service (DDoS) attacks. In these attacks the attacker launches resource exhaustion attacks using the combined power of many hosts to exhaust the resources of a victim. Frequency bandwidth, virtual carrier-sense, association flood, SYN flood, and UDP flood attacks are examples of this category. These examples will be discussed briefly.

SYN Flood Attack: In a legitimate TCP connection establishment, a source host sends a SYN (synchronize/start) packet to a destination host. The destination host responds with a SYN ACK (synchronize acknowledge) packet. The destination host must receive an ACK (acknowledgement) packet before the connection is established. These exchanged packets set up the receive and send windows for each participant. This is known as the "TCP three-way handshake."

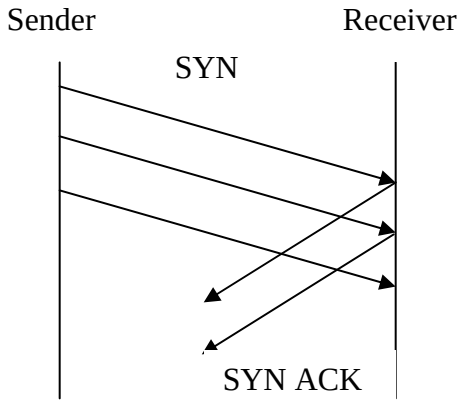


Figure 14. Syn Flood DoS Attack

In this attack type, the attacker sends SYN packets to the victim with the incorrect source IP addresses. The victim machine will send back SYN ACK packets to a nonexistent system in order to complete the three-way handshake. The victim machine will never receive the expected ACK packets. While the victim machine is waiting for the ACK packet, the limited-size connection queue will keep track of all uncompleted connections. The attacker generates SYN packets with fake IP addresses at a rapid rate and fills up the victim's queue. The victim's resources are flooded and consumed by new

connection requests. The flood of malicious SYN packets prevents the victim from responding to legitimate requests. Figure 14 shows the Syn Flood DoS attack.

UDP Flood Attack: The UDP protocol is a connectionless, unreliable protocol which does not require a three-way handshake negotiation between the client and the server. The UDP Flood attack uses UDP echo and character generator (chargen) services. These services generate a series of characters for testing network programs. The attacker forges UDP packets to initiate communications between these services. While they are sending and receiving a flood of useless characters, the two services consume all the bandwidth between the machines.

Frequency Bandwidth Attack: The 802.11 networks use radio signals as the physical medium. Since the medium is shared by all potential users, an attacker who knows the frequency band can launch a DoS attack using very strong radio signals to disrupt the network. The attacker uses a powerful radio transmitter or signal generator to saturate the 802.11 frequency band with noise. Since the Signal-to-Noise Ratio (SNR) reduces to an unusable level, this type of attack will decrease the efficiency of the network or prevent the clients from accessing the network.

The use of powerful radio signals at a close range is a risky attack. If the attacker uses the directional antenna instead of an omni-directional antenna for this attack, then the administrator of the network can easily detect the attacker's location using special network tools, such as AirMagnet.

It is worth noting that cordless phones, microwaves, and many of the Bluetooth devices that use 2.4GHz frequency band may cause this type of DoS attack unintentionally and disrupt 802.1b wireless networks.

Virtual Carrier-Based DoS Attacks: A virtual carrier-sense mechanism is designed to prevent a collision with mobile stations that are not within range of the transmitting station but are within range of the AP. This mechanism was discussed in Chapter II in detail. The control frames are sent in the clear between the clients and APs. The attacker can exploit this vulnerability and mount DoS attacks by spoofing one of these control frames.

The clear-to-send (CTS) DoS attack uses a weakness in the media access control in wireless networks. The CTS frames, sent in response to a request-to-send (RTS) frame, indicate that the channel is free and the receiving end is ready to receive the data packets. The transmitting station sends the RTS after waiting for the random back-off time following the DIFS period. If the medium is still available and the Network Allocation Vector (NAV) reaches zero, then it will send the RTS frame to the AP requesting the reservation of the medium for a period of time. The AP will respond to the clients, including the nodes within the AP's transmission range but beyond the requesting client's range ("hidden nodes"), with the CTS frame announcing the reservation of the medium for a period of time. The nodes that receive RTS or CTS frames must defer their transmissions until after receiving the intended destination's final acknowledgment frame.

An attacker can launch the DoS attack by sending the forged CTS frames at a specific time interval. The stations that receive the CTS frames will update their NAV table with the value in the duration field of the CTS packet received. The attacker will send the CTS frames just before its NAV value reaches zero. The legitimate clients will be blocked even though the medium is available. The attacker does not need a valid MAC address to launch this type of DoS attack. These attacks can be launched using either RTS or ACK frames since they allow the modification of the NAV table using the duration field of these frames.

Distributed Syn Flood Attacks: This attack differs from the SYN Flood by using more than one computer to launch the attack. The main concept for this attack is to create a flood effect on the victim's machine using widely distributed computers. The attacker will install the remote attack programs on weakly protected computers using available hacking tools. Finally, the attacker organizes the attack from these remote computers at the same time.

This huge flood effect consumes the victim's resources and connection bandwidth. The attackers usually launch this type of attack, which requires considerable effort, against big companies' servers. The servers become cut off from the Internet and are incapable of providing services. [7]

2. Software Exploit DoS Attacks

In software exploits, the attacker sends a few packets to exercise specific software bugs within the target's OS or application, thereby disabling the victim. [11] These vulnerabilities are the result of programming or design flaws. When the vulnerable system receives an unexpected packet from the attacker, exceptional conditions occur. The attacker exploits implementation flaws in the service being attacked and makes the victim's system crash by sending a single or a few packets.

The DoS attacks against TCP/IP implementation flaws are also classified in this category. "Land attack," "Ping-of-Death," and many buffer overflow attacks are examples of this category. Since the attack's characteristics are not expected behavior of the legitimate user, implementation-based DoS attacks can easily be detected. This type of DoS attack can be prevented by fixing bugs, applying patches, and filtering the malformed packets.

Teardrop Attack: TCP/IP protocol divides large IP packets into smaller fragments to send over limited capacity lines. The receiving end will reconstruct these fragments according to IP packet header fields, such as the fragment offset and packet ID. All the fragments of the same IP packet carry the same packet ID field.

The Teardrop attack takes advantage of the lack of bounds checking in the whole fragmentation and reassembly process. The attacker sends fragments with offsets that do not align by manipulating a field in TCP/IP packets, called a "fragment offset." This makes reassembly of all the packets impossible since the positions overlap. This vulnerability in the TCP/IP stack cannot be handled properly by certain operating systems. The victim will stop the communication until the system is restarted. It is possible that unsaved data in applications open at the time of attack will be lost. [6]

Land Attack: The Land attack occurs when an attacker sends an SYN packet with the same source and destination IP addresses. In this way, the attacker tries to get the targeted host to establish TCP sessions with itself, making its services unavailable to other legitimate hosts.

Ping of Death Attack: the Ping of Death is an attempt by an attacker to crash or freeze a system by sending an illegal ICMP packet to the victim. The Ping command is used with a small payload to provide a fast, low bandwidth test of connectivity.

The maximum packet size is 65536 bytes in TCP/IP protocol. Packets exceeding this size limit can cause the victim's system to crash. First, the attacker issues a ping command to see if the victim system responses. The ICMP packet is sent in the form of a fragmented message. When the packet is reassembled, the size of the packet will be larger than the maximum legal IP packet size. Some operating systems cannot handle an oversized ICMP echo-request packet. They generate an exception and halt communication on the network. [6]

D. SUMMARY

This chapter presents a classification schema for DoS attacks based on the types of system or protocol weaknesses exploited. This schema includes two main categories: OS/Network and Protocol Weakness DoS attacks. These main categories are divided into sub-categories. The specifications and characteristics of each sub-category of DoS attacks are explained in detail.

The categorization of DoS attacks presented in this chapter will provide a basis for the DoS attack implementation in Chapter V. The various attack implementations based on the categorization schema will be discussed. The security protocols will be evaluated in order to provide a security mechanism for each sub-category DoS attack.

V. ATTACK IMPLEMENTATION AND EVALUATION

A. INTRODUCTION

This chapter describes a set of DoS attack scenarios with a simple tool derived from the hostap access point source code. In addition, this chapter describes a threat model based on the vulnerability taxonomy presented in Chapter IV. The threat model is used to determine the security requirements specific to DoS attacks. These requirements are necessary to secure the WLANs against such attacks. Finally, this chapter will discuss how to enhance the 802.11, the 802.1X and the 802.11i protocols to meet the security requirements critical to preventing or reducing the risk of attack.

B. DENIAL OF SERVICE ATTACK SCENARIOS

A previous thesis, by Selcuk Ozturk [10], demonstrated a DoS attack by spoofing the victim's MAC address and sending a malicious deauthentication packet. In this section, additional DoS attack scenarios will be demonstrated using a simple tool derived from the *hostap* source code. These attacks will prove the weaknesses of the protocols and show the utility of the test-bed. The setup of the test-bed was explained in Chapter III. Therefore, only the infrastructure, modified code, and analysis of the attacks will be discussed in this section.

1. Attack Infrastructure

The attack infrastructure is slightly different from the existing test-bed. A WinXP supplicant with a WLAN 54g W450 embedded wireless card was added to the open-source test-bed while DoS attacks were being created. So, two WinXP supplicants served the role of legitimate clients. A Linux machine running FreeRADIUS was used as the authenticator server. The D-Link DWL 7000 access point was chosen for the role of the legitimate authenticator.

As an attacker system, hostap was used in the access point mode since it provides flexibility for changing the MAC address and modifying the source code. A laptop running the Ethereal packet sniffer with the embedded wireless card was used to verify the attacks. Figure 15 shows the attack infrastructure.

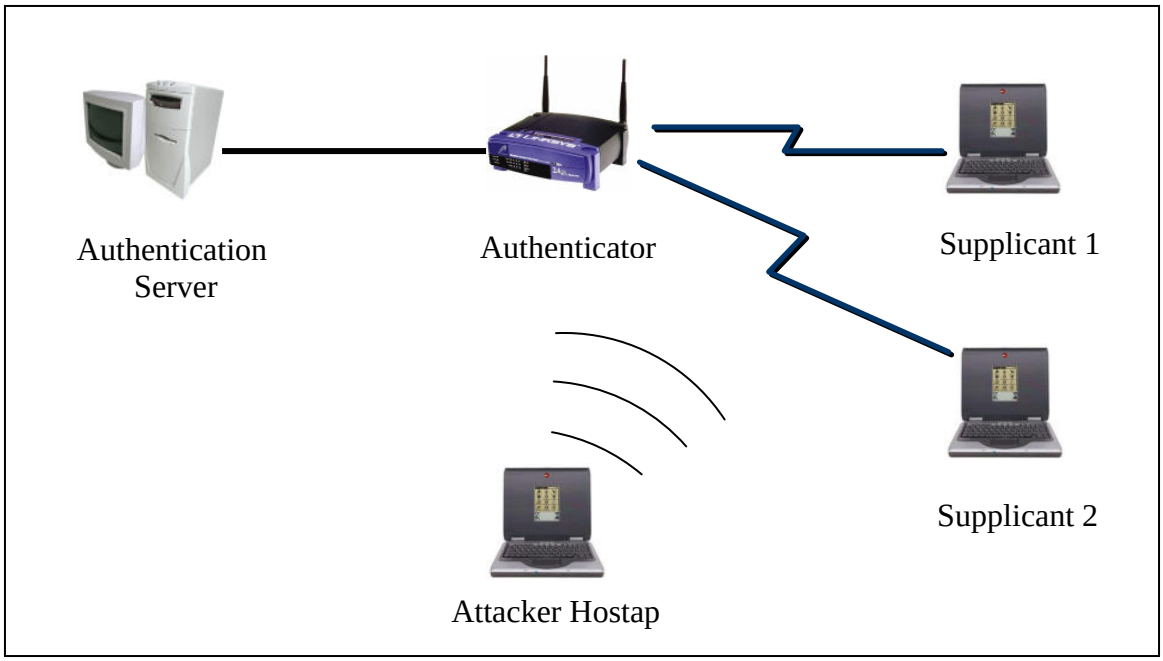


Figure 15. Attack Infrastructure

2. Modification of the Hostap Source Code

The drivers and utilities package that are part of the hostAP tarball are enough to run the Linux machine as a hostap access point. In addition, the hostap daemon must be installed in the open-source test-bed optionally and compiled, along with its drivers, in order to use the counterfeit access point as an attack platform. This daemon will transfer all access point functionalities from the driver level to the user level. Thus, the daemon provides flexibility to modify and manipulate the source code of the hostap access point for the user. The user only needs to recompile the changed C file and re-link the code modules in order for the changes to take effect. It does not require any kernel recompilations and module reinitializations.

The hostap authenticator source code was examined before the manipulation of the code. Most of the functions that are embedded into different C files can be called externally. This feature provides great flexibility for the user to manipulate the code. The "hostapd.c" file includes the main function, which controls all the other files and headers. Table 1 shows important file definitions for the hostap access point source code.

hostapd.c	main daemon which controls other files
config.c	configuration file parsing and data structure definition
driver.c	helper function for configuring hostap kernel driver
sta_info.c	keeps all authenticated supplicant's information
ieee802_11.c	management frame sending and processing file
ieee802_11_auth.c	access control list for 802.11 authentication
ieee802_1x.c	receives and sends EAP and EAPOL packets
eapol_sm.c	keeps track of all clients for 802.1X state machine
accounting.c	sends RADIUS Accounting start and stop messages to the RADIUS
eloop.c	for registering timeout calls, signal handlers and socket read events
radius.c	RADIUS message generation and parsing functions
receive.c	receive all incoming frames from the kernel driver via wlan interface

Table 1. File Definitions in Hostap

There are three different operation modes for hostap. These modes are examined to decide which mode is suitable for the desired attacks. Each of them has tradeoffs. For example, the monitor mode only receives and parses the packets, and the hostap does not transmit any packets in this mode. Since the attacks depend on sending spoofed and malformed packets, the hostap is used in the master mode. But the master mode has disadvantages that show all beacon frames from legitimate access points. It is difficult to follow the program when all beacons are written on the screen. This mode is the default for the hostap software. The following command can be used for changing the hostap mode into the master mode.

iwconfig wlan0 mode Master

Before the implementation of the attacks, the eloop_run( ) function was disabled since the hostap source code would not be used as an access point during the attack phase. The hostap source code is only used as a packet sender. The following four different DoS attacks are implemented by modifying the hostap source code.

- Deauthentication DoS attack
- Disassociation DoS attack

- EAPOL-Logoff DoS attack
- Resource Exhaustion with EAPOL-Start packets

The following C code is inserted into the main function of the hostapd.c file. The infinite “for loop” will prompt the user for the selection of the attack. The user can launch different attacks consecutively.

```
/*removed from original hostap source code to disable incoming beacon
frames*/
//    eloop_run();
      int orh;
      for(;;){

      printf("\n=====================================================");
            printf("\nFor deauthentication DoS attack enter 1:");
            printf("\nFor disassociation DoS attack enter 2:");
            printf("\nFor EAPOL-Logoff DoS attack enter 3:");
            printf("\nFor Resource Exh. EAPOL-Start attack enter 4:");
            printf("\nFor exit enter 5:");
      printf("\n=====================================================");

            /* reading from keyboard*/
            scanf("%d",&orh);

            if(orh==1){
                  u8 addr[ETH_ALEN];
                  memset(addr, 0xff, ETH_ALEN);
                  ieee802_11_send_deauth(&hapd, addr,
                        WLAN_REASON_PREV_AUTH_NOT_VALID);
                  printf("Deauthentication DoS attack launched");
            }

            if(orh==2){
                  u8 addr[ETH_ALEN]={0x00, 0x0b, 0xcd, 0x75, 0x8a, 0x84};
                  ieee802_11_send_disassoc(&hapd, addr,
                        WLAN_REASON_DISASSOC_DUE_TO_INACTIVITY);
                  printf("Disassociation DoS attack launched\n");
            }

            if(orh==3){
                  u8 addr[ETH_ALEN]={0x00, 0x0b, 0xcd, 0x75, 0x8a, 0x84};
                  struct sta_info s;
                  memcpy(s.addr, addr, ETH_ALEN);
                  ieee802_1x_send(&hapd,&s,
                        IEEE802_1X_TYPE_EAPOL_LOGOFF,NULL,0);
                  printf("eapol logoff DoS attack launched\n");
            }

            if(orh==4){
                  int count;
                  u8 addr[ETH_ALEN]={0x00, 0x05, 0x5d, 0x99, 0x60, 0xa6};
```

```
                struct sta_info s;
                memcpy(s.addr, addr, ETH_ALEN);

                      for(count=0;count<100;count++){


                ieee802_1x_send(&hapd,&s,
                      IEEE802_1X_TYPE_EAPOL_START,NULL,0);
                }
                printf("Resource Exh. attack launched\n");
          }

          if(orh==5){
                printf("halt the program\n");
                break;
          }
                printf("please enter the correct value\n")*/

     }/*for loop end*/
```

Table 2. The Modified Hostap Source Code

Before launching the DoS attacks, legitimate supplicants are authenticated with the legitimate access point to represent the normal network connectivity. The ethereal software is used to verify this message exchange. The filter is applied to Ethereal so that it shows only related packets. Figure 16 shows this message sequence.

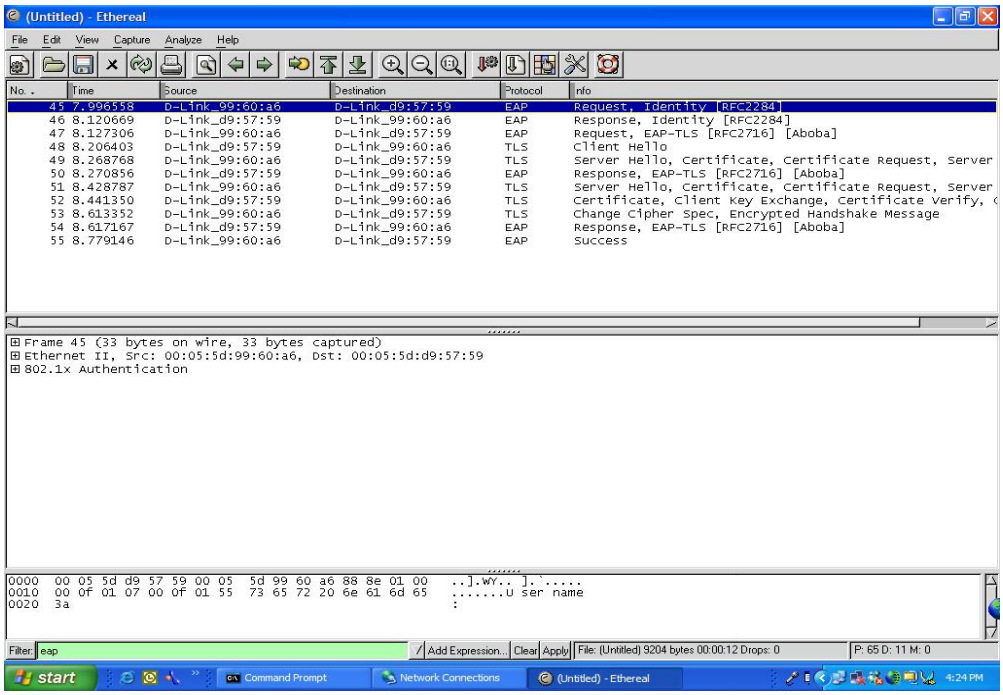


Figure 16. Legitimate Client Authentication

After the legitimate connection of the supplicants, the access point MAC address is spoofed for the first three attacks. The attacker will send the deauthentication, disassociation, and EAPOL-Logoff packets on behalf of the legitimate access point. The following command is used to change the MAC address in the Linux OS and to run the modified code. It is not necessary to spoof the ESSID of WLAN since it is embedded in "hostapd.conf" file. Figure 17 shows the interface of the attacker program.

ifconfig wlan0 hw ether XX:XX:XX:XX:XX:XX

./hostapd –d hostap.conf

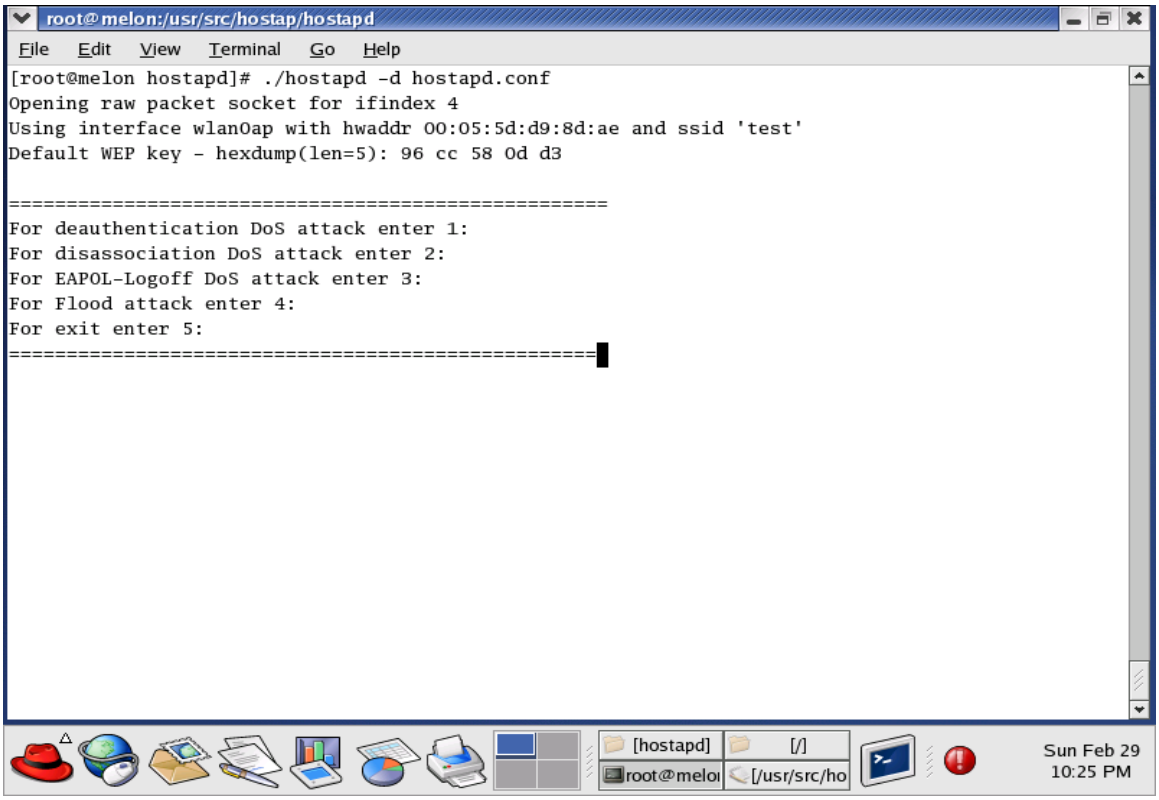


Figure 17. Attacker Program Interface

According to the user interface, respectively, the first, third, and fourth attacks will be discussed in detail since the disassociation DoS attack gives the same result when it is compared to the first attack. Only the function and its parameters are different.

First Attack: The forge deauhentication packet was created for this attack. The function ieee802_11_send_deauth( ) was externally called from the hostapd main function passing the required parameters. The destination MAC address and the reason code of the deauthetication are enough to create this packet. This packet was destined to both legitimate supplicants by using the broadcast address (oxff). After the legitimate supplicants receive these packets, the supplicants are immediately disconnected from the network. They then remain disconnected until the attacker's MAC address is changed from that of the valid access point.

Ozturk's thesis [10] achieved this attack by implicitly running the hostapd source code instead of modifying the source code.

Third Attack: A spoofed EAPOL-Logoff packet was created by passing the required parameters to the ieee802_1x_send( ) function. The parameters include a hostapd object, a sta_info struct, the subtype of the EAPOL frame, the data, and the data length. Since the EAPOL-Logoff frame does not encapsulate an EAP frame, the data and data length must be NULL and zero respectively. The sta_info struct only contains the destination MAC address.

First, the destination MAC address was set to the broadcast address (0xff) but both legitimate clients did not accept this packet, since the packet is not sent to the specific client. So the packet was sent to the MAC address of the Supplicant 2 (00:0B:CD:75:8A:84). The Supplicant 2 was immediately disconnected from the network after receiving the forge EAPOL-Logoff packet and tried to reinitiate the authentication phase. However, the consecutive EAPOL-Logoff DoS attacks prevented the Supplicant 2 from reauthentication.

Fourth Attack: For the EAPOL-Start flood attack, the same ieee802_1x_send( ) method was used by changing the subtype parameter. The aim of this attack is to flood the legitimate access point by sending a huge number of EAPOL-Start packets. Since the EAPOL-Start packet is originally sent after association, one of the associated supplicant's MAC addresses is spoofed.

For each execution of the attack, one hundred bogus EAPOL-Start packets were sent to the legitimate access point. The attack was executed four times at five seconds

intervals so as not to overwhelm the attacker buffer space. After sending four hundred packets, both supplicants were still in the “authentication succeeded” phase. In fact, they could not connect to network resources since the legitimate access point could not handle their request. Figure 18 shows the bogus EAPOL-Start packets captured by the Ethereal packet sniffer.

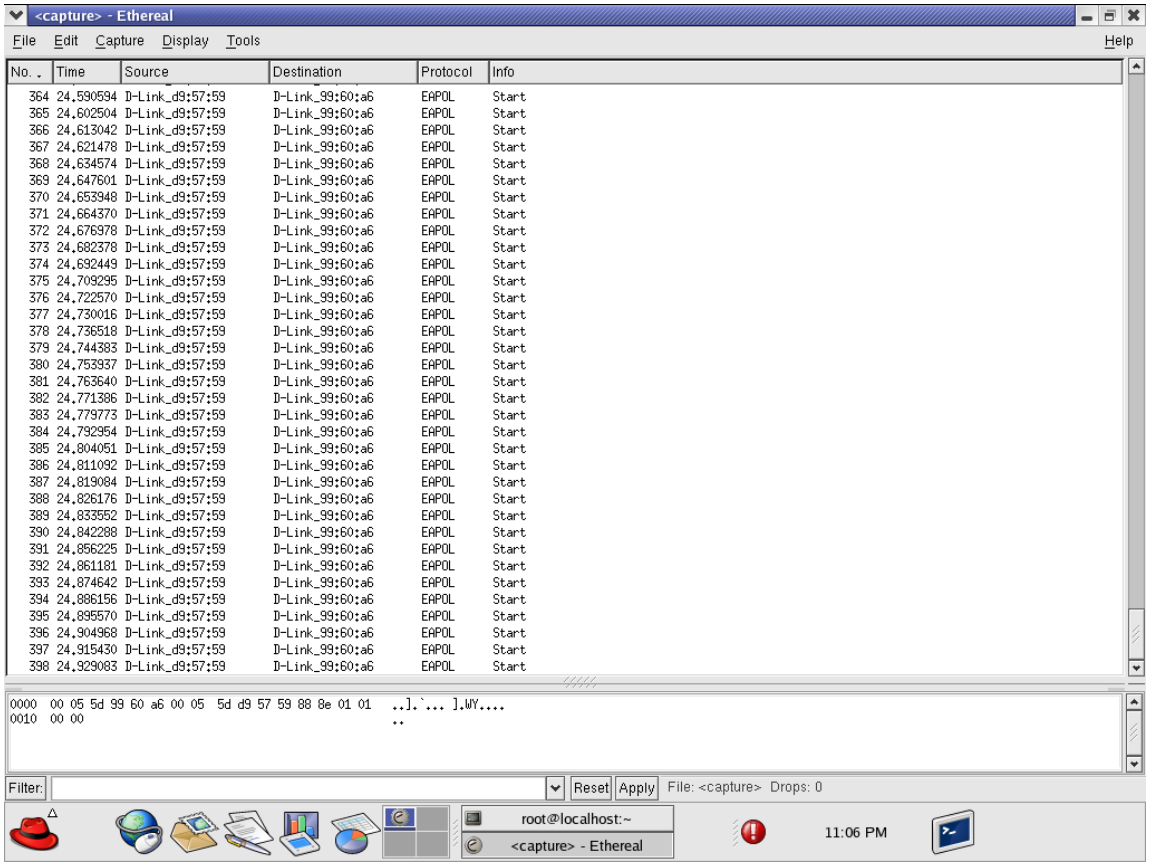


Figure 18. EAPOL-Start Flooding Attack

3. Analysis of the Attacks

The deauthetication and disassociation DoS attacks were successful. After sending only one forge packet, the victims were disconnected from the network. Supplicant 1 gained connectivity after a few seconds, but Supplicant 2 stayed in the “validating identity” phase until the attacker’s MAC address was changed so that it no longer spoofed Supplicant 2. The difference was the result of confusion due to two hosts having the same MAC address.

The result for the EAPOL-Logoff DoS attack was also successful. After the victim was disconnected from the network, the victim reinitiated the EAP-TLS session. However, consecutive bogus packets prevented the victim from gaining the requested network connectivity.

The most effective DoS attack is the EAPOL-Start Flood attack, which overwhelms the access point's buffer space. After receiving 400 forged packets, sent by the attacker, the legitimate access point was unable to respond to any client. In fact, the access point needed to be rebooted after this attack in order to free its buffer space.

This thesis has demonstrated that all protocol specific types of DoS attack can be launched easily against current WLAN protocols. On one hand, the authenticator in the test-bed only allows manipulation of the management frames. So, the deauthentication and disassociation frames for the 802.11 protocol and 802.1X frames must be created by the user. On the other hand, the control frames including RTS, CTS and ACK packets, which are used in virtual carrier-based DoS attacks, cannot be modified by the hostap authenticator in the user level; even though the attacker circumvents the frame fields, the wireless card firmware will overwrite these fields with appropriate values before transmission. These attacks can only be launched with external packet creators such as the Agilent Signal Generator.

C. EVALUATION OF THE DENIAL OF SERVICE ATTACKS

This section will present a threat model and discuss the 802.11, 802.1X and the 802.11i protocols from the perspective of the DoS attacks. The weaknesses and the strengths of these protocols, regarding the DoS attacks, will be evaluated. The DoS attack taxonomy presented in Chapter IV is used to generalize the proposed solutions.

1. Threat Model

A thread model is required to determine whether or not WLANs are vulnerable against DoS attacks. This model will provide a method to analyze and evaluate the security requirements and solutions. The model is described below.

Preconditions:

a) WLANs are widely deployed in companies, schools, governments, and military sectors.

b) Many of the casual or naive users are unaware of the importance of security and most of these WLANs are used without activating the authentication and encryption mechanisms.

c) The 802.11 and 802.1X protocols used in WLANs have vulnerabilities against DoS attacks.

d) Packet sniffers and open-source tools for creating spoofed packets are freely available on the Internet.

Threats:

a) An attacker attempts to flood the victim resources by sending associate response packets.

b) An attacker mounts a DoS attack by sending unauthorized management frames, including deauthentication and disassociation packets.

c) An attacker sends spoofed control frames, including RTS, CTS and packets in order to launch virtual carrier-sense DoS attacks.

d) An attacker sends spoofed EAPOL-Logoff and EAP-Failure packets to launch DoS attacks.

e) An attacker sends malformed or mistimed messages to an AP to crash the AP.

Security Requirements:

a) The protection of control and management frames must be provided with per-packet authentication.

b) WLANs must provide mutual authentication between the access point and client stations.

c) The protocols used in WLANs must be coupled flawlessly to avoid the DoS attacks.

d) AP software must be stress tested.

2. Evaluation of 802.11, 802.1X and 80211i Protocols

Analyzing the strength and weaknesses of these protocols in DoS attacks will determine whether or not the use of a particular protocol is suitable for WLANs. Basically, the 802.1X protocols are designed for use in wired networks. The 802.11i task group recommends using this protocol on top of the existing 802.11 protocol, which is severely flawed. The attacker who mounts DoS attacks against wireless networks usually exploits the holes between these protocols. Merging the 802.11 and 802.1X state machines is problematic.

The 802.11 protocol enables authentication to be performed before the association by allowing the supplicants to authenticate more than one access points while associating with only one. However, the 802.1X authentication occurs after the 802.11 association is established. Figure 19 shows the state machines of these processes.

The supplicant that uses the 802.1X protocol will receive an EAP-Success packet for the successful authentication. Consequently, the authentication server will provide a key with the EAP-Key packet for the confidentiality and integrity of the data packets. It is obvious that the 802.1X protocol does not provide security for management frames. This protocol only provides for the authenticity of the supplicants and the protection of the data packets. In addition, the protocol opens new holes for DoS attacks. The EAPOL-Logoff, EAP-Failure packets that are used by this protocol are subject to spoofing and an attacker can use these types of packets to launch DoS attacks.

According to the state machine of the 802.1X protocol, there are two apparent causes for the vulnerabilities to DoS attacks. One of them is the 802.1X protocol itself and the other one is the loose coupling of the 802.11 and 802.1X protocols. The state machines of both protocols were examined in order to mitigate the effects of the DoS attacks. The following two solutions are proposed against DoS attacks in the 802.1X protocol.

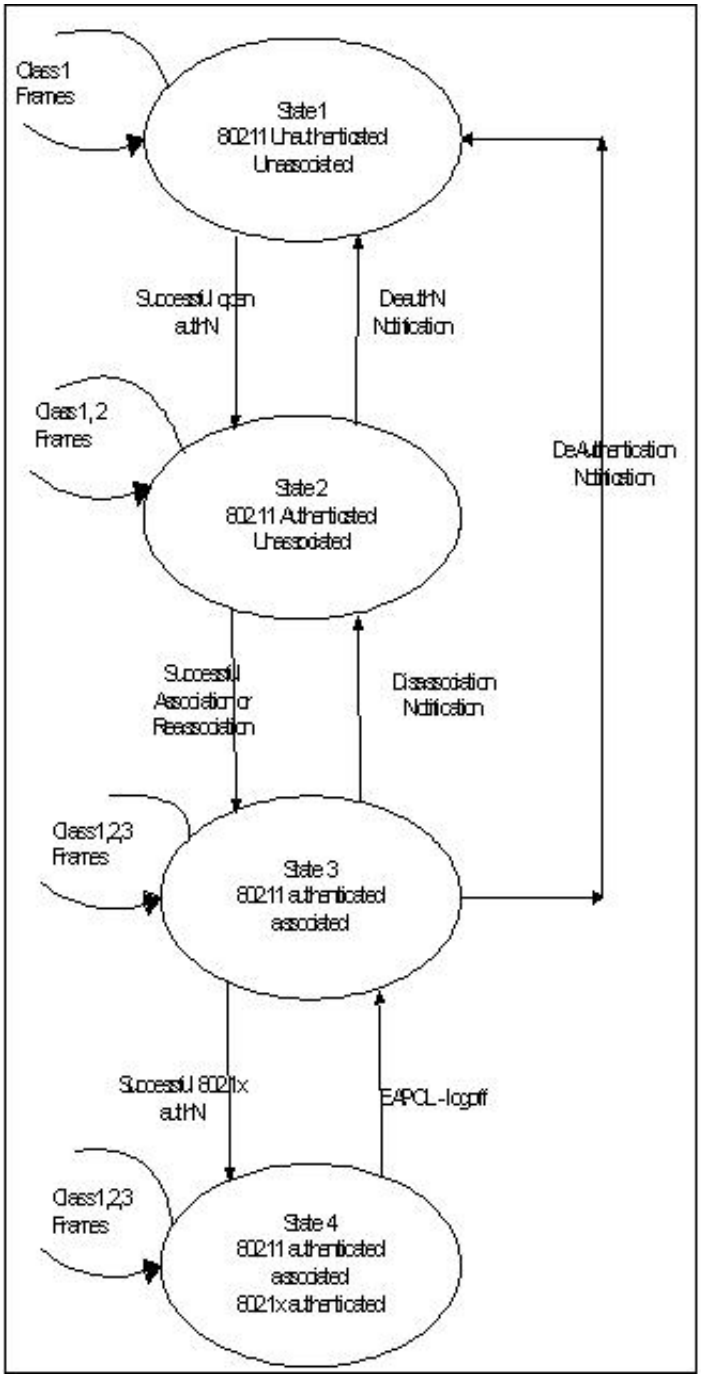


Figure 19. 802.1X State Machine (From Ref. 16)

Discarding the Deauthentication Frames: According to the state machine presented in Figure 19, the supplicant can still receive and process Class 1,2, and 3 frames even though the 802.1X authentication and association are achieved. If the

supplicant receives a deauthentication frame in State 3 and 4, it will immediately return to State 1, which is the “unassociated and unauthenticated” state.

Since the deauthentication of the supplicant is handled by EAPOL-Logoff frames in the 802.1X protocol, it is not necessary to process deauthentication frames in all states, creating a situation where an unexpected deauthentication frame is mishandled. This approach makes State 2 unnecessary, since the authentication process takes place in the upper layer rather than the MAC layer in the 802.1X protocol. The new state machine for this approach is presented in Figure 20.

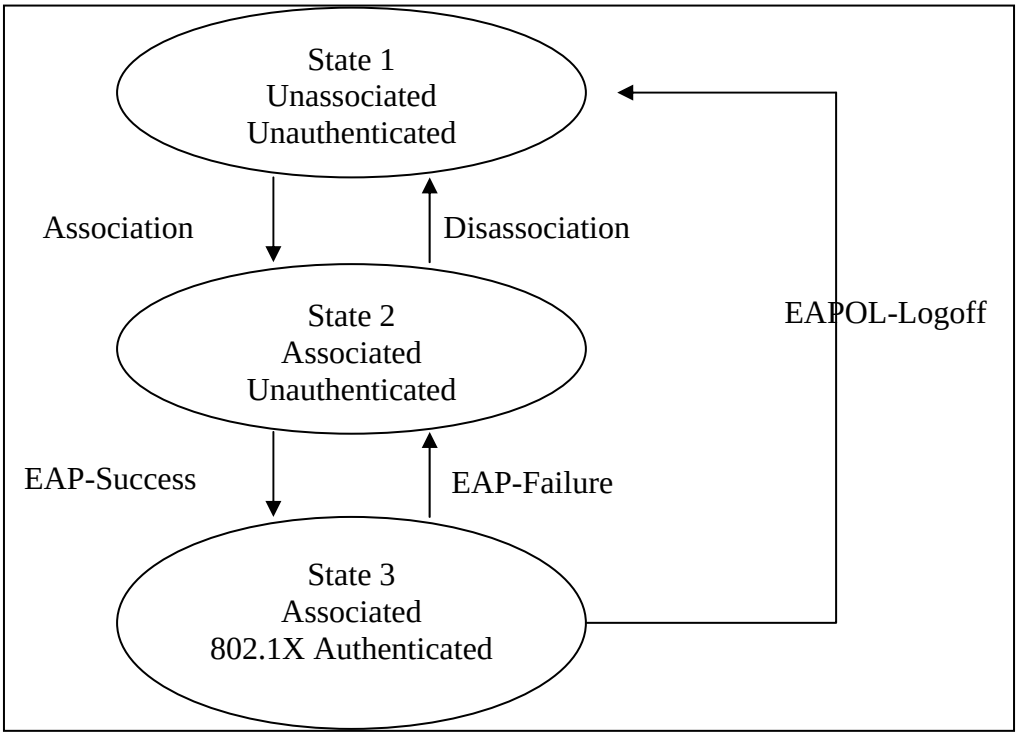


Figure 20. Modified State Machine for 802.1X Protocol

The modified state machine will not allow any authentication frame from the 802.11 protocols. The modification can be applied to the 802.1X protocol by only changing the state variables rather than changing the whole MAC layer. For example, if the supplicant or the access point receives the type “00” and subtype “”1011” or “1100” of the frame indicating the authentication and deauthentication packet respectively, then all the state variables remain unchanged, thus preventing the state transition.

This approach will only solve the deauthentication DoS attack sub-category in the classification framework. However, this approach is still vulnerable to disassociation DoS attacks.

Protecting the Management Frames : Management frames must be authenticated and protected against DoS attacks. Two approaches are possible for authenticating management frames, including Association request/response, Reassociation request/response, Disassociation and Deauthentication.

- Using ciphers such as WEP, TKIP or WRAP to authenticate and to encrypt management frames.
- Authenticator information in management frames

The use of a cipher is more suitable since this process only requires hardware support for cryptographic operations. However, the use of a cipher for protecting the management frames simplifies the key hierarchy since existing keying material is used for encrypting management frames as well as data. If WEP ciphersuite is used to protect the management frames, then the shared secret will be used as a keying material between the supplicant and authenticator. If TKIP or WRAP is used as a ciphersuite, then the 4-way handshake is used to derive a Pairwise Master Key (PMK) as a keying material prior to the exchange of the management frames.

The access point and supplicant capabilities are exchanged during the association phase, so the selection of the ciphersuite is decided in the Association/Reassociation request. The receiving supplicant does not know which ciphersuite is selected before receiving the encrypted management frames. So, the receiving end will decrypt the association request/response frames with all available ciphersuites in order to find which ciphersuite is chosen.

In the 802.1X protocol, the WEP key can be used for encryption of the frame. Since WEP is a shared key it will simplify the key management. However, well-known flaws of WEP make this ciphersuite vulnerable to various types of attacks. So, the 802.11i workgroup proposed the use of the Temporal Key Integrity Protocol (TKIP) as a ciphersuite to avoid the WEP vulnerabilities.

The Message Integrity Check (MIC) must be used to protect the entire management frame, including the frame control, duration, sequence control, and frame bodies. So, the encryption needs to be applied to the MAC Protocol Data Unit (MPDU) to cover these fields.

Another approach proposed by the 802.11i workgroup to protect the management frames is the addition of authenticator information to these frames. The ciphersuite and MIC were originally used for the protection of the data frames. A new field in the frame format can be defined as the authenticator information for the pertinent part of the frames. This approach will eliminate the negotiation of the cipersuite prior to sending authenticating management frames.

The authenticator information element includes algorithm type, MIC, replay counter, and length. The algorithm supported by the 802.11i protocol is HMAC-SHA1. This algorithm calculates the MIC of the desired field of the frame. When the peer supplicant receives the frame, it will calculate the MIC to ensure the authenticity and integrity of the management frame.

Protecting the management frames will secure the WLANs against protocol-based DoS attacks in the classification schema. These two approaches can be applied to control frames to protect the integrity and authenticity of the packets to secure the WLANs against virtual carrier-sense DoS attacks.

D. SUMMARY

In this chapter, four different DoS applications are presented, each executed by modifying the hostap source code. The structure of the hostap C files and relations between these files are discussed to provide a direction for future research.

A threat model is defined to determine the security requirements of the WLANs for protection against DoS attacks. The strength and the weaknesses of the protocols are evaluated and solutions are proposed for DoS attacks in the classification schema provided. The 802.1X protocol designed on top of the 802.11 protocol does not add protection against DoS attacks. Rather, it is shown that this protocol opens new holes for DoS attacks.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION AND FUTURE WORK

A. CONCLUSION

This thesis primarily examined the Denial of Service attacks in Wireless networks. First, the vulnerabilities and the components of the existing 802.11 and 802.1X wireless protocols were introduced briefly.

For verification and analysis of DoS attacks against WLANs, an 802.1X test-bed was successfully built on an IEEE 802.11 wireless LAN. The thesis explained in detail how to build and configure the 802.1X entities: the supplicant, the authenticator and the authentication server. The open-source software was used on the Linux Operating System environment for availability and ease of source-code manipulation. This test-bed provided a basis for the implementation of the DoS attacks in WLANs.

This thesis provided a classification framework to categorize all DoS attacks in both wired and wireless networks. The DoS attacks were classified into two main categories: attacks on the host operating system (OS) or network infrastructure and attacks that exploit specific weaknesses of the target protocol. These main classes were branched into the sub-categories according to the flaws exploited by the attacker.

Since it is difficult to find a solution to many different types of DoS attack, this classification framework helped us to generalize the solutions for each category. When a new DoS attack is encountered, the specification of this attack can easily be identified with this schema.

The four different DoS applications are presented by modifying the hostap source code. The deauthentication DoS attack, the disassociation DoS attack, the EAPOL-Logoff DoS attack and EAPOL-Start flood DoS attack are successfully launched against the legitimate clients that use the 802.1X protocol. These attacks proved that the 802.1X protocol designed on top of 802.11 protocol does not add any protection against DoS attacks, on the other hand this protocol opens new holes for DoS attacks.

A threat model is defined to determine the security requirements of the WLANs for DoS attacks. The strength and the weaknesses of the existing protocols are evaluated and solution approaches are proposed against the DoS attacks to meet the security

requirements of the WLANs. These approaches will mitigate the effects of DoS attacks or completely secure the WLANs for a specific sub-category of the DoS attacks defined in classification schema.

B. FUTURE WORK

This thesis proved that malformed packet creation is easy with hostap source code. In addition, the incoming packets can be analyzed by modifying the parse function in the hostap source code. With the implementation of this packet analyzer, an Intrusion Detection System (IDS) for DoS attacks can be implemented with the Graphical User Interface (GUI).

The IDS will analyze the packet fields and warn the system administrator when the attack conditions are met. For example, consecutive EAPOL-Start frames from the same MAC address can be considered as an attack condition. If malformed packets are captured, then the system administrator can take precautions by discarding the malformed packets. This work requires experience in C programming language and the Linux OS environment.

APPENDIX A

A. CERTIFICATE GENERATOR CONFIGURATION

1. OpenSSL Configuration File

```
#
# OpenSSL example configuration file.
# This is mostly being used for generation of certificate requests.
#

# This definition stops the following lines choking if HOME isn't
# defined.
HOME            = .
RANDFILE        = $ENV::HOME/.rnd

# Extra OBJECT IDENTIFIER info:
#oid_file       = $ENV::HOME/.oid
oid_section     = new_oids

# To use this configuration file with the "-extfile" option of the
# "openssl x509" utility, name here the section containing the
# X.509v3 extensions to use:
# extensions            =
# (Alternatively, use a configuration file that has only
# X.509v3 extensions in its main [= default] section.)

[ new_oids ]

# We can add new OIDs in here for use by 'ca' and 'req'.
# Add a simple OID like this:
# testoid1=1.2.3.4
# Or use config file substitution like this:
# testoid2=${testoid1}.5.6

####################################################################
[ ca ]
default_ca  = CA_default            # The default ca section

####################################################################
[ CA_default ]

dir         = ./demoCA        # Where everything is kept
certs       = $dir/certs            # Where the issued certs are kept
crl_dir         = $dir/crl        # Where the issued crl are kept
database    = $dir/index.txt  # database index file.
new_certs_dir     = $dir/newcerts         # default place for new
certs.

certificate = $dir/cacert.pem       # The CA certificate
serial          = $dir/serial           # The current serial number
crl         = $dir/crl.pem          # The current CRL
private_key = $dir/private/cakey.pem # The private key
```

```
RANDFILE    = $dir/private/.rand    # private random number file

x509_extensions   = usr_cert        # The extentions to add to the cert

# Comment out the following two lines for the "traditional"
# (and highly broken) format.
# name_opt  = ca_default            # Subject Name options
# cert_opt  = ca_default            # Certificate field options

# Extension copying option: use with caution.
# copy_extensions = copy

# Extensions to add to a CRL. Note: Netscape communicator chokes on V2
CRLs
# so this is commented out by default to leave a V1 CRL.
# crl_extensions  = crl_ext

default_days      = 365             # how long to certify for
default_crl_days= 30                # how long before next CRL
default_md  = md5             # which md to use.
preserve    = no              # keep passed DN ordering

# A few difference way of specifying how similar the request should
look
# For type CA, the listed attributes must be the same, and the optional
# and supplied fields are just that :-)
policy            = policy_match

# For the CA policy
[ policy_match ]
countryName       = match
stateOrProvinceName     = match
organizationName  = match
organizationalUnitName  = optional
commonName        = supplied
emailAddress            = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName       = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName  = optional
organizationalUnitName  = optional
commonName        = supplied
emailAddress            = optional

####################################################################
[ req ]
default_bits            = 1024
default_keyfile   = privkey.pem
distinguished_name      = req_distinguished_name
attributes        = req_attributes
x509_extensions   = v3_ca     # The extentions to add to the self
signed cert
```

```
# Passwords for private keys if not present they will be prompted for
# input_password = secret
# output_password = secret

# This sets a mask for permitted string types. There are several
options.
# default: PrintableString, T61String, BMPString.
# pkix       : PrintableString, BMPString.
# utf8only: only UTF8Strings.
# nombstr : PrintableString, T61String (no BMPStrings or UTF8Strings).
# MASK:XXXX a literal mask value.
# WARNING: current versions of Netscape crash on BMPStrings or
UTF8Strings
# so use this option with caution!
string_mask = nombstr

# req_extensions = v3_req # The extensions to add to a certificate
request

[ req_distinguished_name ]
countryName                     = Country Name (2 letter code)
countryName_default             = US
countryName_min                 = 2
countryName_max                 = 2

stateOrProvinceName             = State or Province Name (full name)
stateOrProvinceName_default     = California

localityName                    = Locality Name (eg, city)
localityName_default            = Monterey

0.organizationName              = Organization Name (eg, company)
0.organizationName_default      = NPGS

# we can do this but it is not needed normally :-)
#1.organizationName             = Second Organization Name (eg, company)
#1.organizationName_default     = World Wide Web Pty Ltd

organizationalUnitName          = Organizational Unit Name (eg, section)
organizationalUnitName_default      = SAAM

commonName                      = Common Name (eg, YOUR name)
commonName_max                  = 64
commanName_default              = WirelessSAAM CA

emailAddress                    = Email Address
emailAddress_max                = 64
emailAddress_default            = oozan@nps.navy.mil

# SET-ex3                       = SET extension number 3

[ req_attributes ]
challengePassword               = A challenge password
challengePassword_min           = 4
challengePassword_max           = 20
```

```
unstructuredName        = An optional company name

[ usr_cert ]

# These extensions are added when 'ca' signs a request.

# This goes against PKIX guidelines but some CAs do it and some
software
# requires this to avoid interpreting an end user certificate as a CA.

basicConstraints=CA:FALSE

# Here are some examples of the usage of nsCertType. If it is omitted
# the certificate can be used for anything *except* object signing.

# This is OK for an SSL server.
# nsCertType                    = server

# For an object signing certificate this would be used.
# nsCertType = objsign

# For normal client use this is typical
# nsCertType = client, email

# and for everything including object signing:
# nsCertType = client, email, objsign

# This is typical in keyUsage for a client certificate.
# keyUsage = nonRepudiation, digitalSignature, keyEncipherment

# This will be displayed in Netscape's comment listbox.
nsComment               = "OpenSSL Generated Certificate"

# PKIX recommendations harmless if included in all certificates.
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer:always

# This stuff is for subjectAltName and issuerAltname.
# Import the email address.
# subjectAltName=email:copy
# An alternative to produce certificates that aren't
# deprecated according to PKIX.
# subjectAltName=email:move

# Copy subject details
# issuerAltName=issuer:copy

#nsCaRevocationUrl              = http://www.domain.dom/ca-crl.pem
#nsBaseUrl
#nsRevocationUrl
#nsRenewalUrl
#nsCaPolicyUrl
#nsSslServerName

[ v3_req ]

# Extensions to add to a certificate request
```

```
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment

[ v3_ca ]

# Extensions for a typical CA

# PKIX recommendation.

subjectKeyIdentifier=hash

authorityKeyIdentifier=keyid:always,issuer:always

# This is what PKIX recommends but some broken software chokes on
critical
# extensions.
#basicConstraints = critical,CA:true
# So we do this instead.
basicConstraints = CA:true

# Key usage: this is typical for a CA certificate. However since it
will
# prevent it being used as an test self-signed certificate it is best
# left out by default.
# keyUsage = cRLSign, keyCertSign

# Some might want this also
# nsCertType = sslCA, emailCA

# Include email address in subject alt name: another PKIX
recommendation
# subjectAltName=email:copy
# Copy issuer details
# issuerAltName=issuer:copy

# DER hex encoding of an extension: beware experts only!
# obj=DER:02:03
# Where 'obj' is a standard or added object
# You can even override a supported extension:
# basicConstraints= critical, DER:30:03:01:01:FF

[ crl_ext ]

# CRL extensions.
# Only issuerAltName and authorityKeyIdentifier make any sense in a
CRL.

# issuerAltName=issuer:copy
authorityKeyIdentifier=keyid:always,issuer:always
```

B. CERTIFICATE GENERATION SCRIPTS

1. Root Certificate Authority Generation Script

```
#!/bin/sh
SSL=/usr/local/openssl-certgen
export PATH=${SSL}/bin/:${SSL}/ssl/misc:${PATH}
export LD_LIBRARY_PATH=${SSL}/lib
# needed if you need to start from scratch otherwise the CA.pl -newca
command doesn't copy the new
# private key into the CA directories
rm -rf demoCA
echo
"**********************************************************************
**********"
echo "Creating self-signed private key and certificate"
echo "When prompted override the default value for the Common Name
field"
echo
"**********************************************************************
**********"
echo
# Generate a new self-signed certificate.
# After invocation, newreq.pem will contain a private key and
certificate
# newreq.pem will be used in the next step
openssl req -new -x509 -keyout newreq.pem -out newreq.pem -passin
pass:whatever -passout pass:whatever
echo
"**********************************************************************
**********"
echo "Creating a new CA hierarchy (used later by the "ca" command) with
the certificate"
echo "and private key created in the last step"
echo
"**********************************************************************
**********"
echo
echo "newreq.pem" | CA.pl -newca >/dev/null
echo
"**********************************************************************
**********"
echo "Creating ROOT CA"
echo
"**********************************************************************
**********"
echo
# Create a PKCS#12 file, using the previously created CA
certificate/key
# The certificate in demoCA/cacert.pem is the same as in newreq.pem.
Instead of
# using "-in demoCA/cacert.pem" we could have used "-in newreq.pem" and
then omitted
# the "-inkey newreq.pem" because newreq.pem contains both the private
key and certificate
```

```
openssl pkcs12 -export -in demoCA/cacert.pem -inkey newreq.pem -out
root.p12 -cacerts -passin pass:whatever -passout pass:whatever
# parse the PKCS#12 file just created and produce a PEM format
certificate and key in root.pem
openssl pkcs12 -in root.p12 -out root.pem -passin pass:whatever -
passout pass:whatever
# Convert root certificate from PEM format to DER format
openssl x509 -inform PEM -outform DER -in root.pem -out root.der
#Clean Up
rm -rf newreq.pem
```

2. Server Certificate Generation Script

```
#!/bin/sh
SSL=/usr/local/openssl-certgen
export PATH=${SSL}/bin/:${SSL}/ssl/misc:${PATH}
export LD_LIBRARY_PATH=${SSL}/lib
echo
"**********************************************************************
***********"
echo "Creating server private key and certificate"
echo "When prompted enter the server name in the Common Name field."
echo
"**********************************************************************
***********"
echo
# Request a new PKCS#10 certificate.
# First, newreq.pem will be overwritten with the new certificate
request
openssl req -new -keyout newreq.pem -out newreq.pem -passin
pass:whatever -passout pass:whatever
# Sign the certificate request. The policy is defined in the
openssl.cnf file.
# The request generated in the previous step is specified with the -
infiles option and
# the output is in newcert.pem
# The -extensions option is necessary to add the OID for the extended
key for server authentication
openssl ca -policy policy_anything -out newcert.pem -passin
pass:whatever -key whatever -extensions xpserver_ext -extfile
xpextensions -infiles newreq.pem
# Create a PKCS#12 file from the new certificate and its private key
found in newreq.pem
# and place in file specified on the command line
openssl pkcs12 -export -in newcert.pem -inkey newreq.pem -out $1.p12 -
clcerts -passin pass:whatever -passout pass:whatever
# parse the PKCS#12 file just created and produce a PEM format
certificate and key in certsrv.pem
openssl pkcs12 -in $1.p12 -out $1.pem -passin pass:whatever -passout
pass:whatever
# Convert certificate from PEM format to DER format
openssl x509 -inform PEM -outform DER -in $1.pem -out $1.der
# Clean Up
rm -rf newert.pem newreq.pem
```

3. Supplicant Certificate Generation Script

```
#!/bin/sh
SSL=/usr/local/openssl-certgen
export PATH=${SSL}/bin/:${SSL}/ssl/misc:${PATH}
export LD_LIBRARY_PATH=${SSL}/lib
echo
"**********************************************************************
***********"
echo "Creating client private key and certificate"
echo "When prompted enter the client name in the Common Name field.
This is the same"
echo " used as the Username in FreeRADIUS"
echo
"**********************************************************************
***********"
echo
# Request a new PKCS#10 certificate.
# First, newreq.pem will be overwritten with the new certificate
request
openssl req -new -keyout newreq.pem -out newreq.pem -passin
pass:whatever -passout pass:whatever
# Sign the certificate request. The policy is defined in the
openssl.cnf file.
# The request generated in the previous step is specified with the -
infiles option and
# the output is in newcert.pem
# The -extensions option is necessary to add the OID for the extended
key for client authentication
openssl ca -policy policy_anything -out newcert.pem -passin
pass:whatever -key whatever -extensions xpclient_ext -extfile
xpextensions -infiles newreq.pem
# Create a PKCS#12 file from the new certificate and its private key
found in newreq.pem
# and place in file specified on the command line
openssl pkcs12 -export -in newcert.pem -inkey newreq.pem -out $1.p12 -
clcerts -passin pass:whatever -passout pass:whatever
# parse the PKCS#12 file just created and produce a PEM format
certificate and key in certclt.pem
openssl pkcs12 -in $1.p12 -out $1.pem -passin pass:whatever -passout
pass:whatever
# Convert certificate from PEM format to DER format
openssl x509 -inform PEM -outform DER -in $1.pem -out $1.der
# clean up
rm -rf newcert newreq.pem
```

4. XP Specific Extension Files

```
[xpclient_ext]
extendedKeyUsage = 1.3.6.1.5.5.7.3.2
[xpserver_ext]
extendedKeyUsage = 1.3.6.1.5.5.7.3.1
```

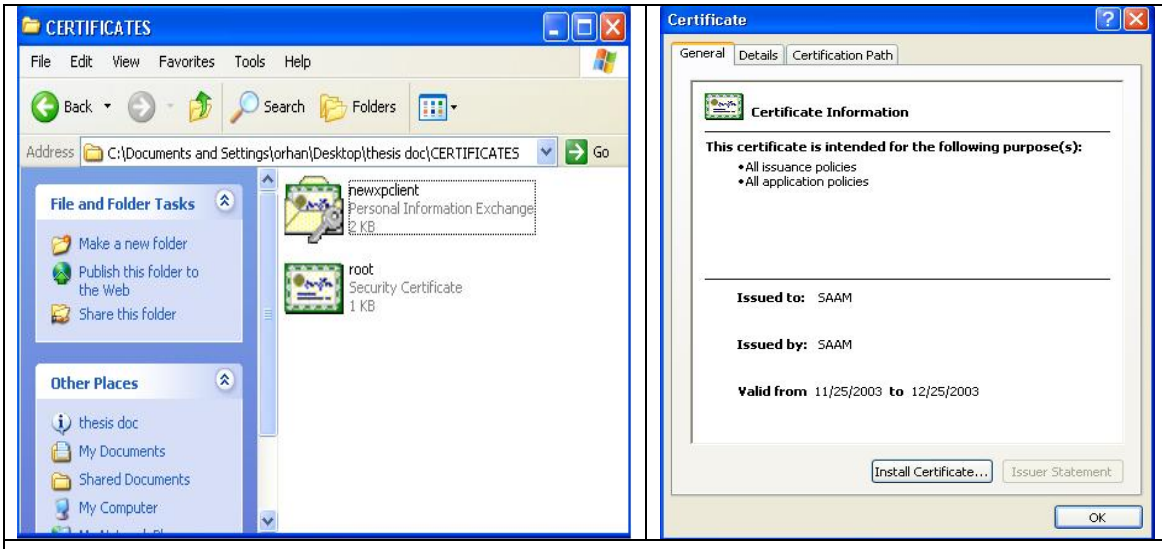
APPENDIX B

A. WINDOWS XP CERTIFICATE INSTALLATION

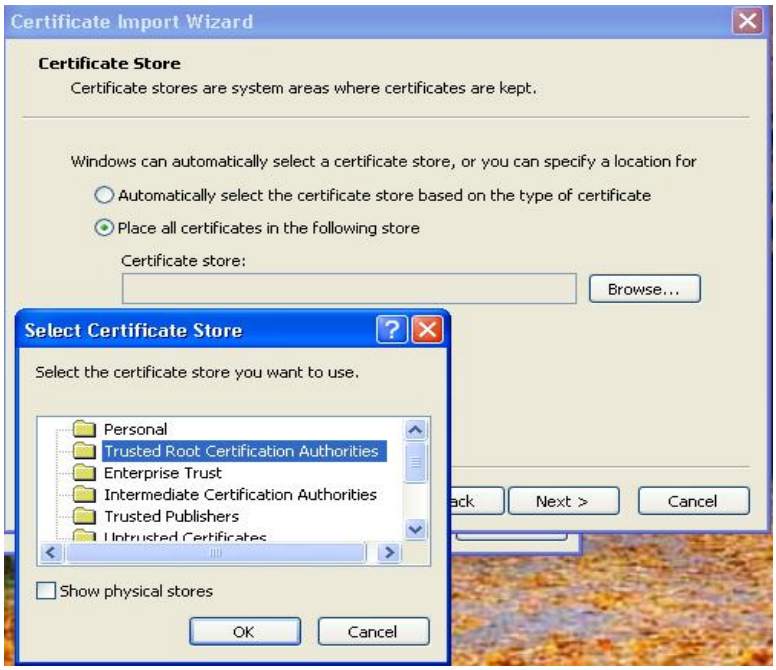
Windows XP with Service Pack 1 support requires a client public key certificate, a private key corresponding to this public key and a rootCA public key certificate to verify the server's certificate authentication.

The "newxpclient.p12" file contains the public key certificate and private key of the client. The "root.der" contains the public key certificate (PKC) of the root certificate authority. We must assure that both of these files are generated in a trusted environment and there is a strong trust relationship between the client and the root certificate authority. First, the rootCA' PKC must be installed manually.

The screen shot demonstration below will explain how to install these certificates:



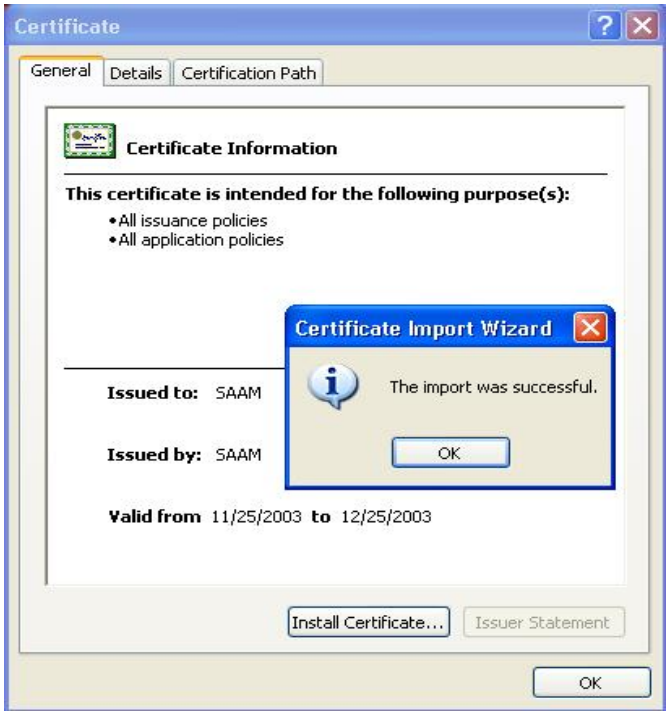
B.A1. When the root.der file is double-clicked, the certificate information window comes up. Then the "Install Certificate" button must be clicked.



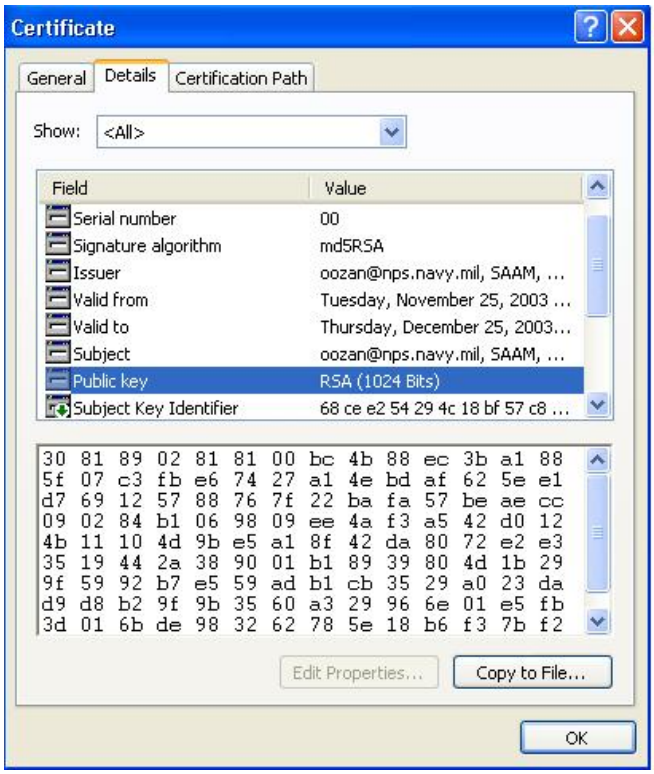
B.A2. The “Place all certificate in the following store” radio button must be checked and the “Trusted Root Certification Authorities” must be highlighted. Then click OK.

B.A3. Click Next to continue

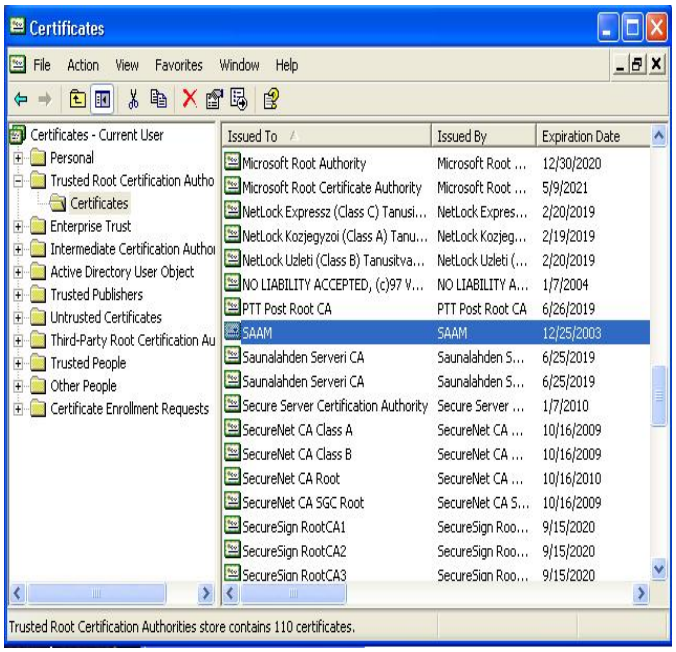
B.A4. Click “Finish” to complete the root certificate import wizard



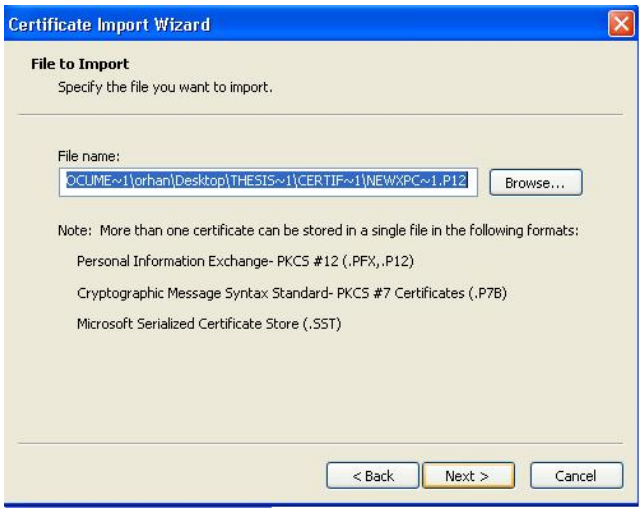
B.A5. If everything has been right. The system responds with the “successful import window



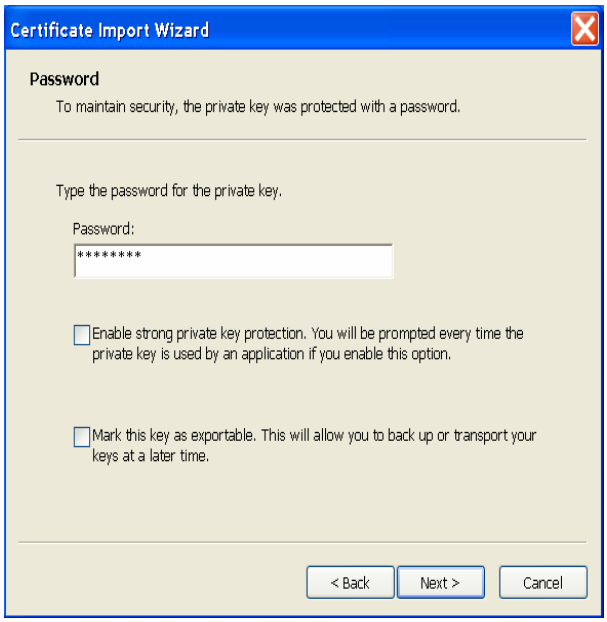
B.A6. 1024 bits RSA public key and other properties of the root certificate can be monitored through the “details” tab



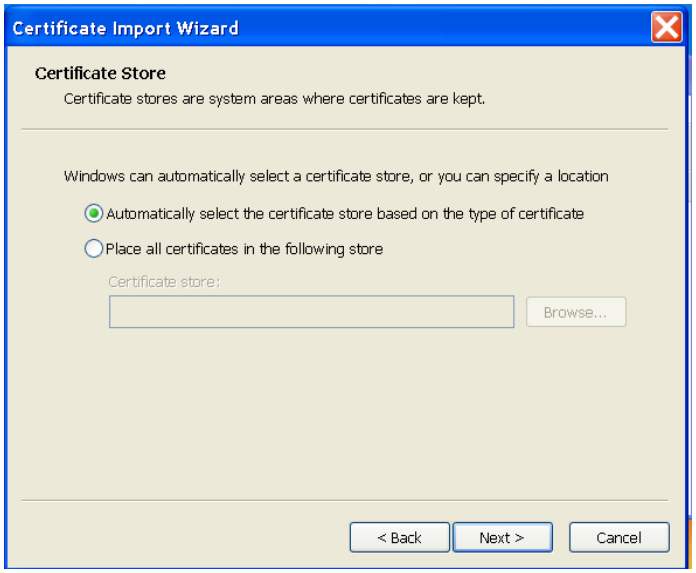
B.A7. The rootCA PKC can be verified via the MMC console whether it is under Trusted Certificate Authorities or not.



B.A8. When the newxpclient.p12 file is double-clicked, the certificate installation wizard appears on the screen. Click “Next” to continue.

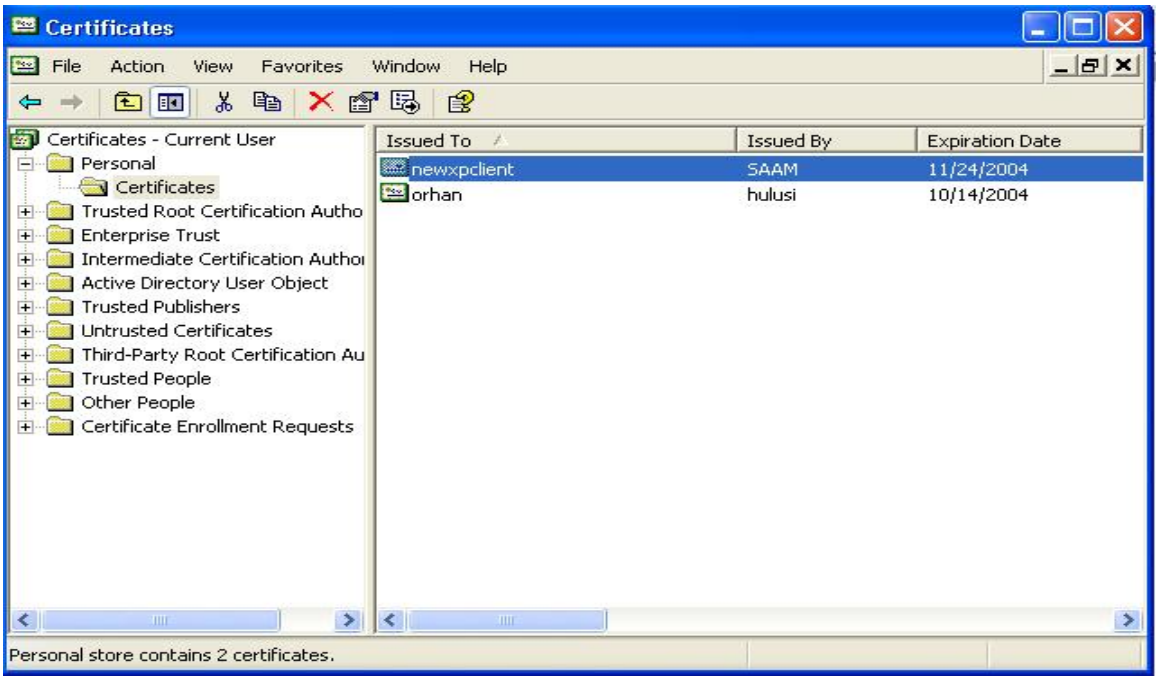


B.A9. The password to decrypt the private key for the client must be entered correctly. The password can be obtained manually from the certificate manager. Then click “Next” to continue.



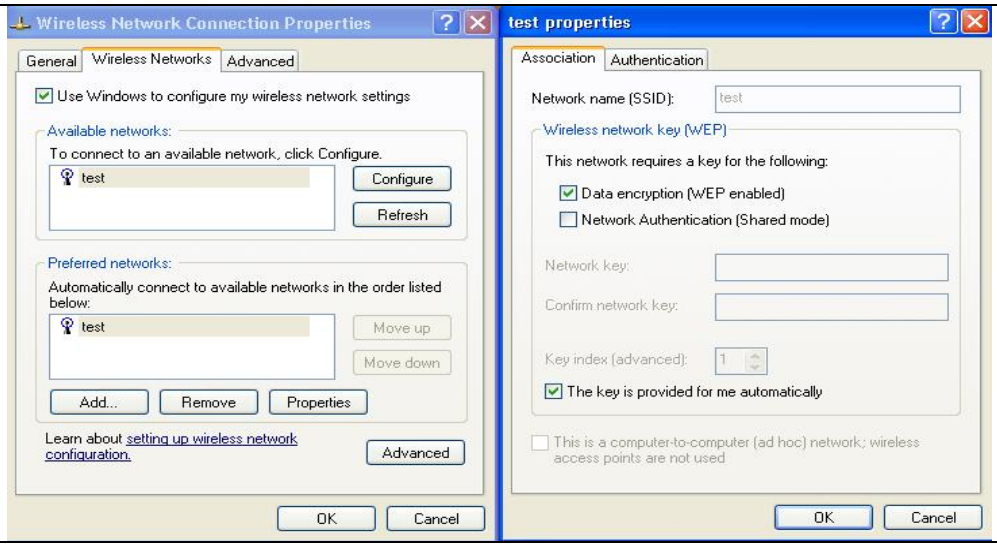
B.A10 Leave the default values and click next

B.A11. Click “Finish” for successful client certificate import

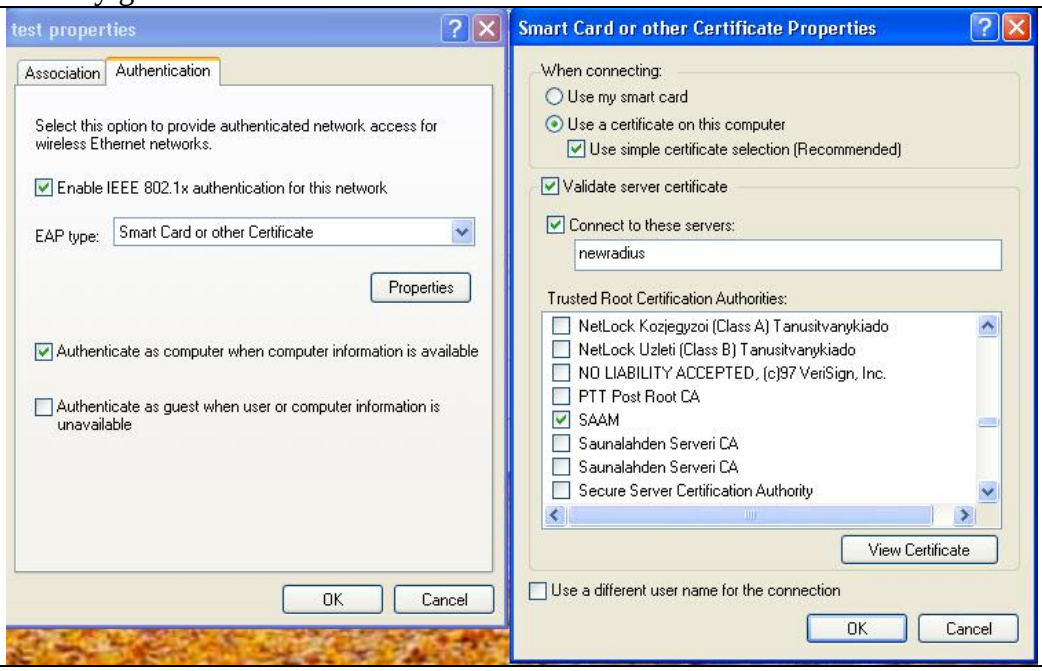


B.A12. The certification path should be verified under in the MMC window.

B. WINDOWS XP WIRELESS CLIENT 802.1X CONFIGURATION



B.B1. The WEP and dynamic key options must be checked in order to support the dynamic key generation from the authenticator.

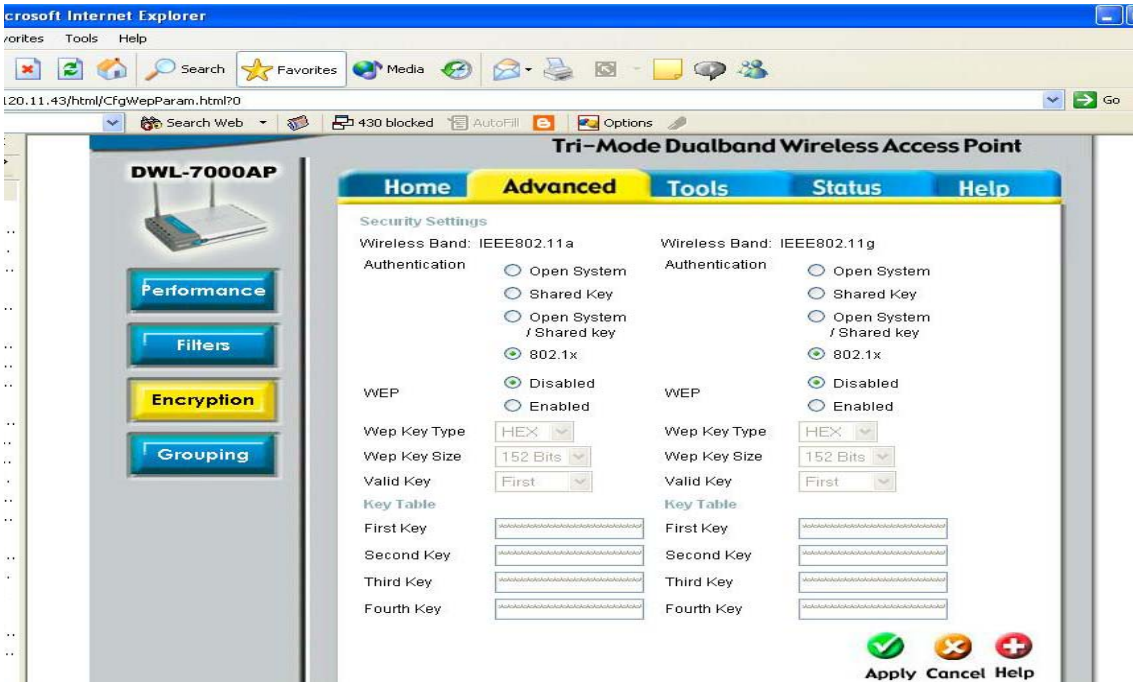


B.B2 The “Enable IEEE 802.1X authentication for this network” radio button must be checked with the “Smart Card or other Certificates” option. When the “Properties” button is clicked, the “Use a certificate on this computer” radio button must be checked. The “Validate server certificate” radio button must be checked; otherwise, only the client certificate is validated at the server. The server certificate must also be validated to enforce mutual authentication.

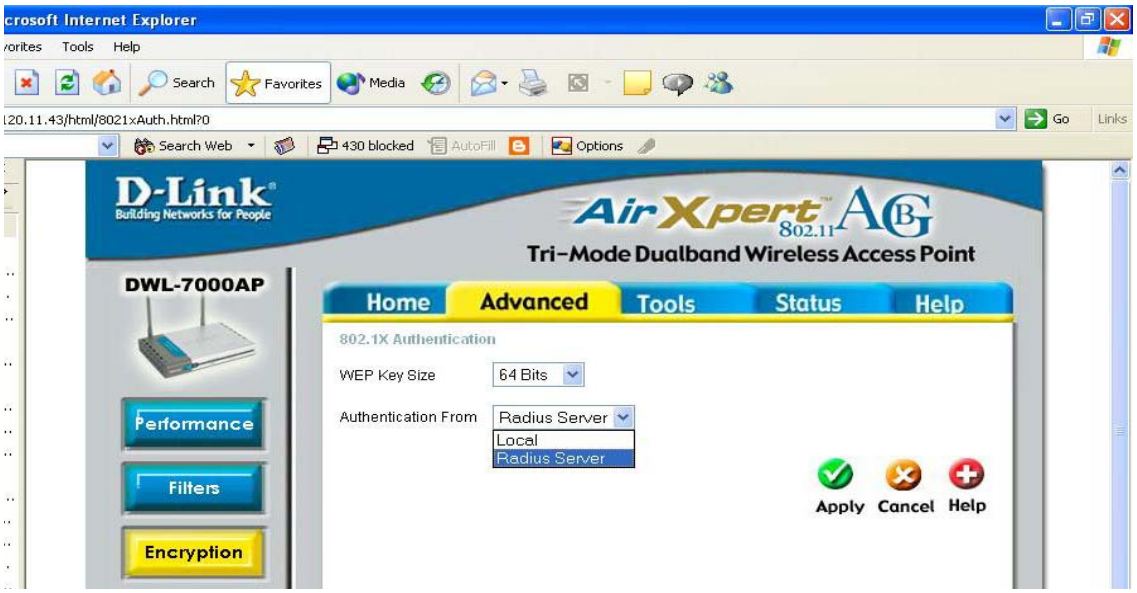
APPENDIX C

A. D-LINK DWL-7000AP CONFIGURATION

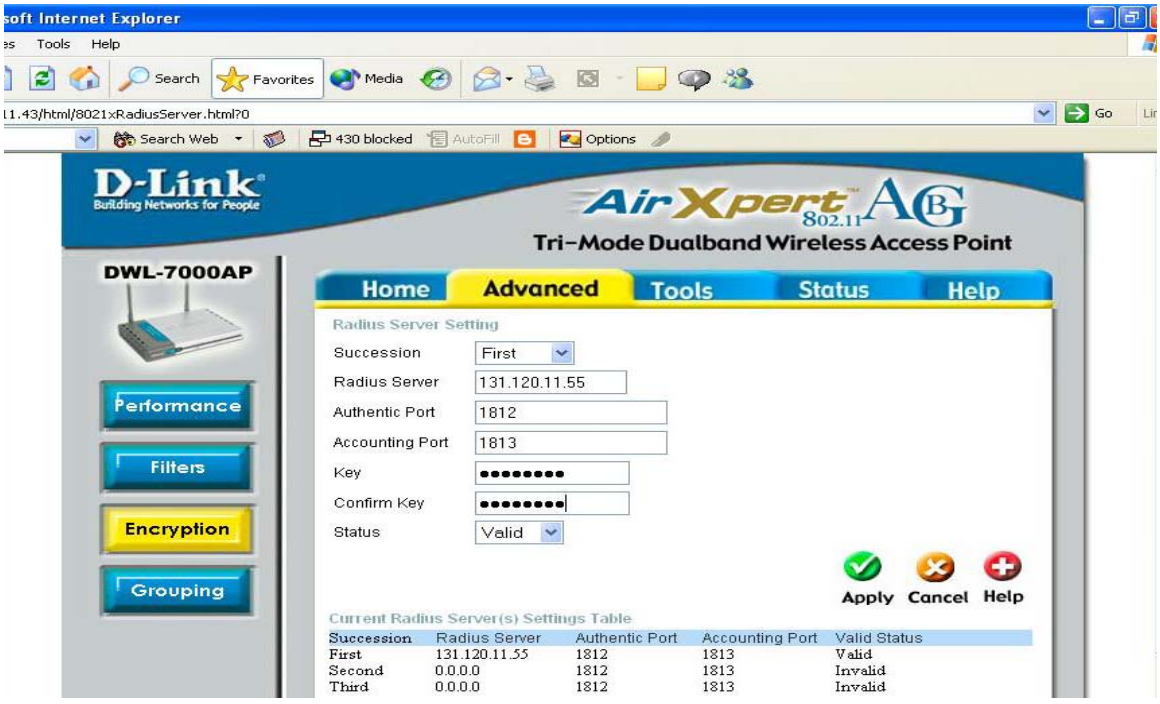
C.A1 D-link DWL-7000AP comes with an access point manager. This manager is useful to identify and discover the access points in the network. After discovering the access point, a new IP address can be assigned.



C.A2 After assigning a legitimate IP address to the access point, a web browser is used to configure the access point. The IP address of the AP is written to the address window and the web-based configuration tool is ready to be used. 802.1X option can be selected under *advanced* tab and *encryption* page.



C.A3 The next page after selecting the 802.1X option is the authentication server selection page. Radius server is selected from the drop down menu.



C.A4 the next page after selecting the radius server helps the user apply the specifications of the authentication server e.g. IP address, authentication port and the shared Key. Once this fields are completed, the AP is restarted and able to serve as an authenticator.

B. HOSTAP CONFIGURATION FILE

```
##### hostapd configuration file
##############################################
# Empty lines and lines starting with # are ignored

# AP netdevice name (without 'ap' prefix, i.e., wlan0 uses wlan0ap for
# management frames)
interface=wlan0

# Debugging: 0 = no, 1 = minimal, 2 = verbose, 3 = msg dumps
debug=3

# Dump file for state information (on SIGUSR1)
dump_file=/tmp/hostapd.dump

# Daemonize hostapd process (i.e., fork to background)
daemonize=1

##### IEEE 802.11 related configuration
#######################################

# SSID to be used in IEEE 802.11 management frames
ssid=test

# Station MAC address -based authentication
# 0 = accept unless in deny list
# 1 = deny unless in accept list
# 2 = use external RADIUS server (accept/deny lists are searched first)
macaddr_acl=0

# Accept/deny lists are read from separate files (containing list of
# MAC addresses, one per line). Use absolute path name to make sure
# that the files can be read on SIGHUP configuration reloads.
#accept_mac_file=/etc/hostapd.accept
#deny_mac_file=/etc/hostapd.deny

# Associate as a station to another AP while still acting as an AP on
# the same channel.
#assoc_ap_addr=00:12:34:56:78:9a


##### IEEE 802.1X (and IEEE 802.1aa/D4) related configuration #################

# Require IEEE 802.1X authorization
ieee8021x=1
```

```
# Use internal minimal EAP Authentication Server for testing IEEE 802.1X.
# This should only be used for testing since it authorizes all users
# that suppot IEEE 802.1X without any keys or certificates.
minimal_eap=0

# Optional displayable message sent with EAP Request-Identity
eap_message=hello

# WEP rekeying (disabled if key lengths are not set or are set to 0)
# Key lengths for default/broadcast and individual/unicast keys:
# 5 = 40-bit WEP (also known as 64-bit WEP with 40 secret bits)
# 13 = 104-bit WEP (also known as 128-bit WEP with 104 secret bits)
wep_key_len_broadcast=5
wep_key_len_unicast=5
#Rekeying period in seconds. 0 = do not rekey (i.e., set keys only once)
wep_rekey_period=300

# EAPOL-Key index workaround (set bit7) for WinXP Supplicant (needed only if
# only broadcast keys are used)
eapol_key_index_workaround=1

##### IEEE 802.11f - Inter-Access Point Protocol (IAPP) ######################

# Interface to be used for IAPP broadcast packets
#iapp_interface=eth0

##### RADIUS configuration
#####################################################
# for IEEE 802.1X with external Authentication Server, IEEE 802.11
# authentication with external ACL for MAC addresses, and accounting

# The own IP address of the access point (used as NAS-IP-Address)
own_ip_addr=131.120.8.145

# RADIUS authentication server
auth_server_addr=131.120.11.55
auth_server_port=1812
auth_server_shared_secret=besiktas

# RADIUS accounting server
#acct_server_addr=127.0.0.1
#acct_server_port=1813
#acct_server_shared_secret=secret
```

APPENDIX D

A. FREERADIUS EAP-TLS MODULE MAKE FILE

```
# Generated automatically from Makefile.in by configure.
TARGET      = rlm_eap_tls
SRCS        = rlm_eap_tls.c eap_tls.c cb.c tls.c mppe_keys.c
RLM_CFLAGS  = $(INCLTDL) -I../..    -DOPENSSL_NO_KRB5
HEADERS     = rlm_eap_tls.h eap_tls.h ../../eap.h ../../rlm_eap.h
RLM_INSTALL =
RLM_LIBS    +=  -lcrypto -lssl

$(STATIC_OBJS): $(HEADERS)

$(DYNAMIC_OBJS): $(HEADERS)

RLM_DIR=../../
include ${RLM_DIR}../rules.mak
```

B. RADIUSD CONFIGURATION FILE

```
##
## radiusd.conf          -- FreeRADIUS server configuration file.
##
##       http://www.freeradius.org/
##       $Id: radiusd.conf.in,v 1.160 2003/10/23 15:38:24 aland Exp $
##


prefix = /usr/local
exec_prefix = ${prefix}
sysconfdir = /etc
localstatedir = ${prefix}/var
sbindir = ${exec_prefix}/sbin
logdir = ${localstatedir}/log/radius
raddbdir = ${sysconfdir}/raddb
radacctdir = ${logdir}/radacct

#  Location of config and logfiles.
confdir = ${raddbdir}
run_dir = ${localstatedir}/run/radiusd

#
#  The logging messages for the server are appended to the
#  tail of this file.
```

```
#
log_file = ${logdir}/radius.log

#
# libdir: Where to find the rlm_* modules.
#
#   This should be automatically set at configuration time.
#
libdir = ${exec_prefix}/lib

#  pidfile: Where to place the PID of the RADIUS server.
#
#  The server may be signalled while it's running by using this
#  file.
#
pidfile = ${run_dir}/radiusd.pid


# user/group: The name (or #number) of the user/group to run radiusd as.
#
#   If these are commented out, the server will run as the user/group
#   that started it.  In order to change to a different user/group, you
#   MUST be root ( or have root privleges ) to start the server.
#
#user = nobody
#group = nobody

#  max_request_time: The maximum time (in seconds) to handle a request.
#
#  Requests which take more time than this to process may be killed, and
#  a REJECT message is returned.
#
#  Useful range of values: 5 to 120
#
max_request_time = 30

#  delete_blocked_requests: If the request takes MORE THAN 'max_request_time'
#  to be handled, then maybe the server should delete it.
#
delete_blocked_requests = no

#  cleanup_delay: The time to wait (in seconds) before cleaning up
#  a reply which was sent to the NAS.
#
cleanup_delay = 5
```

```
#  max_requests: The maximum number of requests which the server keeps
#  track of.  This should be 256 multiplied by the number of clients.
#
max_requests = 1024

#  bind_address:  Make the server listen on a particular IP address, and
#  send replies out from that address.  This directive is most useful
#  for machines with multiple IP addresses on one interface.
#
bind_address = *

#  port: Allows you to bind FreeRADIUS to a specific port.
#
port = 0

#  hostname_lookups: Log the names of clients or just their IP addresses
#  e.g., www.freeradius.org (on) or 206.47.27.232 (off).
#
hostname_lookups = no

#  Core dumps are a bad thing.  This should only be set to 'yes'
#  if you're debugging a problem with the server.
#
#  allowed values: {no, yes}
#
allow_core_dumps = no

#  Regular expressions
#  These items are set at configure time.  If they're set to "yes",
#  then setting them to "no" turns off regular expression support.
#
regular_expressions    = yes
extended_expressions = yes

#  Log the full User-Name attribute, as it was found in the request.
#
# allowed values: {no, yes}
#
log_stripped_names = no

#  Log authentication requests to the log file.
#
#  allowed values: {no, yes}
#
log_auth = no
```

```
#  Log passwords with the authentication requests.
#  allowed values: {no, yes}
#
log_auth_badpass = no
log_auth_goodpass = no

# usercollide:  Turn "username collision" code on and off.  See the
# "doc/duplicate-users" file
#
usercollide = no

# lower_user / lower_pass:
# Lower case the username/password "before" or "after"
# attempting to authenticate.
#
lower_user = no
lower_pass = no

# nospace_user / nospace_pass:
#
#  Some users like to enter spaces in their username or password
#  incorrectly.  To save yourself the tech support call, you can
#  eliminate those spaces here:
#
nospace_user = no
nospace_pass = no

#  The program to execute to do concurrency checks.
checkrad = ${sbindir}/checkrad

# SECURITY CONFIGURATION
#
#  There may be multiple methods of attacking on the server.  This
#  section holds the configuration items which minimize the impact
#  of those attacks
#
security {
        #
        #  max_attributes: The maximum number of attributes
        #  permitted in a RADIUS packet.  Packets which have MORE
        #  than this number of attributes in them will be dropped.
        #
        max_attributes = 200

        #
        #  delayed_reject: When sending an Access-Reject, it can be
```

```
        #  delayed for a few seconds.  This may help slow down a DoS
        #  attack.  It also helps to slow down people trying to brute-force
        #  crack a users password.
        #
        reject_delay = 1

        #
        #  status_server: Whether or not the server will respond
        #  to Status-Server requests.
        #
        status_server = no
}

# PROXY CONFIGURATION
#
#  proxy_requests: Turns proxying of RADIUS requests on or off.
#
proxy_requests  = yes
$INCLUDE  ${confdir}/proxy.conf


# CLIENTS CONFIGURATION
#
#  Client configuration is defined in "clients.conf".
#
$INCLUDE  ${confdir}/clients.conf


# SNMP CONFIGURATION
#
#  Snmp configuration is only valid if SNMP support was enabled
#  at compile time.
#
snmp   = no
$INCLUDE  ${confdir}/snmp.conf


# THREAD POOL CONFIGURATION
#
#  The thread pool is a long-lived group of threads which
#  take turns (round-robin) handling any incoming requests.
#
thread pool {
        #  Number of servers to start initially --- should be a reasonable
        #  ballpark figure.
        start_servers = 5
```

```
        #  Limit on the total number of servers running.
        #
        #  If this limit is ever reached, clients will be LOCKED OUT, so it
        #  should NOT BE SET TOO LOW.  It is intended mainly as a brake to
        #  keep a runaway server from taking the system with it as it spirals
        #  down...
        #
        max_servers = 32

        #  Server-pool size regulation.  Rather than making you guess
        #  how many servers you need, FreeRADIUS dynamically adapts to
        #  the load it sees, that is, it tries to maintain enough
        #  servers to handle the current load, plus a few spare
        #  servers to handle transient load spikes.
        #
        min_spare_servers = 3
        max_spare_servers = 10

        #  There may be memory leaks or resource allocation problems with
        #  the server.  If so, set this value to 300 or so, so that the
        #  resources will be cleaned up periodically.
        #
        max_requests_per_server = 0
}

# MODULE CONFIGURATION
#
#  The names and configuration of each module is located in this section.
#
#  After the modules are defined here, they may be referred to by name,
#  in other sections of this configuration file.
#
modules {
        #
        #  Each module has a configuration as follows:
        #
        #       name [ instance ] {
        #               config_item = value
        #               ...
        #       }
        #
        #  The 'name' is used to load the 'rlm_name' library
        #  which implements the functionality of the module.
        #
        #  The 'instance' is optional.  To have two different instances
```

```
#  of a module, it first must be referred to by 'name'.
#  The different copies of the module are then created by
#  inventing two 'instance' names, e.g. 'instance1' and 'instance2'
#
#  The instance names can then be used in later configuration
#  INSTEAD of the original 'name'.  See the 'radutmp' configuration
#  below for an example.
#

# PAP module to authenticate users based on their stored password
#
pap {
        encryption_scheme = crypt
}

# CHAP module
#
#  To authenticate requests containing a CHAP-Password attribute.
#
chap {
        authtype = CHAP
}

# Pluggable Authentication Modules
#
pam {
        #
        #  The name to use for PAM authentication.
        pam_auth = radiusd
}

# Unix /etc/passwd style authentication
#
unix {
        #
        #  Cache /etc/passwd, /etc/shadow, and /etc/group
        #
        #  The default is to NOT cache them.
        #
        cache = yes

        # Reload the cache every 600 seconds (10mins). 0 to disable.

        cache_reload = 600

        #
```

```
        #  Define the locations of the normal passwd, shadow, and
        #  group files.
        #
                passwd = /etc/passwd
                shadow = /etc/shadow
                group = /etc/group


        #
        #  Where the 'wtmp' file is located.
        #  This should be moved to it's own module soon.
        #
        radwtmp = ${logdir}/radwtmp
}

#  Extensible Authentication Protocol
#
#  For all EAP related authentications
#
#
eap {
        #  Invoke the default supported EAP type when
        #  EAP-Identity response is received.
        #
        default_eap_type = tls

        #  Default expiry time to clean the EAP list, It is
        #  maintained to correlate the EAP-Response for each
        #  EAP-request sent.
        timer_expire     = 60

        #  There are many EAP types, but the server has support
        #  for only a limited subset.  If the server receives
        #  a request for an EAP type it does not support, then
        #  it normally rejects the request.

        ignore_unknown_eap_types = no

        ## EAP-TLS is highly experimental EAP-Type at the moment.
        #       Please give feedback on the mailing list.
        tls {
                private_key_password = whatever
                private_key_file = /etc/1x/newradius.pem

                #  If Private key & Certificate are located in
                #  the same file, then private_key_file &
```

```
                # certificate_file must contain the same file
                # name.
                certificate_file = /etc/1x/newradius.pem

                # Trusted Root CA list
                CA_file = /etc/1x/root.pem

                dh_file = /etc/1x/DH
                random_file = /etc/1x/random

                #
                # This can never exceed the size of a RADIUS
                # packet (4096 bytes), and is preferably half
                # that, to accomodate other attributes in
                # RADIUS packet.  On most APs the MAX packet
                # length is configured between 1500 - 1600
                # In these cases, fragment size should be
                # 1024 or less.
                #
                fragment_size = 1024

                # include_length is a flag which is
                # by default set to yes If set to
                # yes, Total Length of the message is
                # included in EVERY packet we send.
                # If set to no, Total Length of the
                # message is included ONLY in the
                # First packet of a fragment series.
                #
                include_length = yes

                # Check the Certificate Revocation List
                #
                #       check_crl = yes
        }

        mschapv2 {
        }
}

# Microsoft CHAP authentication
#
# This module supports MS-CHAP and MS-CHAPv2 authentication.
# It also enforces the SMB-Account-Ctrl attribute.
#
mschap {
```

```
        #
        #  As of 0.9, the mschap module does NOT support
        #  reading from /etc/smbpasswd.

        authtype = MS-CHAP

}

# Lightweight Directory Access Protocol (LDAP)
#
ldap {
        server = "ldap.your.domain"
        # identity = "cn=admin,o=My Org,c=UA"
        # password = mypass
        basedn = "o=My Org,c=UA"
        filter = "(uid=%{Stripped-User-Name:-%{User-Name}})"
        # base_filter = "(objectclass=radiusprofile)"

        start_tls = no

        # default_profile = "cn=radprofile,ou=dialup,o=My Org,c=UA"
        # profile_attribute = "radiusProfileDn"
        access_attr = "dialupAccess"

        # Mapping of RADIUS dictionary attributes to LDAP
        # directory attributes.
        dictionary_mapping = ${raddbdir}/ldap.attrmap

        ldap_connections_number = 5

        # groupmembership_attribute = radiusGroupName
        timeout = 4
        timelimit = 3
        net_timeout = 1
        # compare_check_items = yes
        # access_attr_used_for_allow = yes
}

# Realm module, for proxying.
#
#  You can have multiple instances of the realm module to
#  support multiple realm syntaxs at the same time.  The
#  search order is defined the order in the authorize and
#  preacct blocks after the module config block.
#
realm realmslash {
```

```
        format = prefix
        delimiter = "/"
}

#  'username@realm'
#
realm suffix {
        format = suffix
        delimiter = "@"
}

#  'username%realm'
#
realm realmpercent {
        format = suffix
        delimiter = "%"
}

# Preprocess the incoming RADIUS request, before handing it off
# to other modules.
#

preprocess {
        huntgroups = ${confdir}/huntgroups
        hints = ${confdir}/hints

        # This hack changes Ascend's wierd port numberings
        # to standard 0-??? port numbers so that the "+" works
        # for IP address assignments.
        with_ascend_hack = no
        ascend_channels_per_line = 23

        # Windows NT machines often authenticate themselves as
        # NT_DOMAIN\username
        #
        with_ntdomain_hack = no

        # Specialix Jetstream 8500 24 port access server.
        #
        with_specialix_jetstream_hack = no

        # Cisco sends it's VSA attributes with the attribute
        # name *again* in the string, like:
        #
        with_cisco_vsa_hack = no
}
```

```
        # Livingston-style 'users' file
        #
        files {
                usersfile = ${confdir}/users
                acctusersfile = ${confdir}/acct_users
                compat = no
        }

        # Write a detailed log of all accounting records received.
        #
        detail {
                #  Note that we do NOT use NAS-IP-Address here, as
                #  that attribute MAY BE from the originating NAS, and
                #  NOT from the proxy which actually sent us the
                #  request.  The Client-IP-Address attribute is ALWAYS
                #  the address of the client which sent us the
                #  request.
                #
                detailfile = ${radacctdir}/%{Client-IP-Address}/detail-%Y%m%d

                #
                #  The Unix-style permissions on the 'detail' file.
                #
                detailperm = 0600
        }

        # Create a unique accounting session Id.  Many NASes re-use or
        # repeat values for Acct-Session-Id, causing no end of
        # confusion.
        #
        acct_unique {
                key = "User-Name, Acct-Session-Id, NAS-IP-Address, Client-IP-Address,
NAS-Port-Id"
        }


        #  Include another file that has the SQL-related configuration.
        #  This is another file only because it tends to be big.
        #
        $INCLUDE  ${confdir}/sql.conf

        #  Write a 'utmp' style file, of which users are currently
        #  logged in, and where they've logged in from.
        #
        radutmp {
```

```
        #  Where the file is stored.  It's not a log file,
        #  so it doesn't need rotating.
        #
        filename = ${logdir}/radutmp

        #  The field in the packet to key on for the
        #  'user' name,  If you have other fields which you want
        #  to use to key on to control Simultaneous-Use,
        #  then you can use them here.
        #
        username = %{User-Name}

        #  Whether or not we want to treat "user" the same
        #  as "USER", or "User".
        case_sensitive = yes

        #  Accounting information may be lost, so the user MAY
        #  have logged off of the NAS, but we haven't noticed.
        #  If so, we can verify this information with the NAS,
        #
        check_with_nas = yes

        # Set the file permissions, as the contents of this file
        # are usually private.
        perm = 0600

        callerid = "yes"
}

# "Safe" radutmp - does not contain caller ID, so it can be
# world-readable, and radwho can work for normal users, without
# exposing any information that isn't already exposed by who(1).
radutmp sradutmp {
        filename = ${logdir}/sradutmp
        perm = 0644
        callerid = "no"
}

# attr_filter - filters the attributes received in replies from
# proxied servers, to make sure we send back to our RADIUS client
# only allowed attributes.
attr_filter {
        attrsfile = ${confdir}/attrs
}

#  counter module:
```

```
#  This module takes an attribute (count-attribute).
#  It also takes a key, and creates a counter for each unique
#  key.  The count is incremented when accounting packets are
#  received by the server.  The value of the increment depends
#  on the attribute type.
#
#  The module should be added in the instantiate, authorize and
#  accounting sections.  Make sure that in the authorize
#  section it comes after any module which sets the
#  'check-name' attribute.
#
counter daily {
        filename = ${raddbdir}/db.daily
        key = User-Name
        count-attribute = Acct-Session-Time
        reset = daily
        counter-name = Daily-Session-Time
        check-name = Max-Daily-Session
        allowed-servicetype = Framed-User
        cache-size = 5000
}

# The "always" module is here for debugging purposes. Each
# instance simply returns the same result, always, without
# doing anything.
always fail {
        rcode = fail
}
always reject {
        rcode = reject
}
always ok {
        rcode = ok
        simulcount = 0
        mpp = no
}

#
#  The 'expression' module currently has no configuration.
expr {
}

#
#  The 'digest' module currently has no configuration.
#
#  "Digest" authentication against a Cisco SIP server.
```

```
#  See 'doc/rfc/draft-sterman-aaa-sip-00.txt' for details
#  on performing digest authentication for Cisco SIP servers.
#
digest {
}

#
#  Execute external programs
#
exec {
        wait = yes
        input_pairs = request
}

#
#  This is a more general example of the execute module.
#
exec echo {
        #
        #  Wait for the program to finish.
        #
        wait = yes

        #
        #  The name of the program to execute, and it's
        #  arguments.  Dynamic translation is done on this
        #  field, so things like the following example will
        #  work.
        #
        program = "/bin/echo %{User-Name}"

        input_pairs = request

        #
        #  Where to place the output attributes (if any) from
        #  the executed program.  The values allowed, and the
        #  restrictions as to availability, are the same as
        #  for the input_pairs.
        #
        output_pairs = reply

}

#  Do server side ip pool management. Should be added in post-auth and
#  accounting sections.
#
```

```
        ippool main_pool {

                #  range-start,range-stop: The start and end ip
                #  addresses for the ip pool
                range-start = 192.168.1.1
                range-stop = 192.168.3.254

                #  netmask: The network mask used for the ip's
                netmask = 255.255.255.0

                #  cache-size: The gdbm cache size for the db
                #  files.
                cache-size = 800

                # session-db: The main db file used to allocate ip's to clients
                session-db = ${raddbdir}/db.ippool

                # ip-index: Helper db index file used in multilink
                ip-index = ${raddbdir}/db.ipindex

                # override: Will this ippool override a Framed-IP-Address already set
                override = no
        }
}

# Instantiation
#
#  This section orders the loading of the modules.  Modules
#  listed here will get loaded BEFORE the later sections like
#  authorize, authenticate, etc. get examined.

instantiate {
        #
        #  The expression module doesn't do authorization,
        #  authentication, or accounting.  It only does dynamic
        #  translation, of the form:
        expr

        #
        # We add the counter module here so that it registers
        # the check-name attribute before any module which sets
        # it
#       daily
}

#  Authorization. First preprocess (hints and huntgroups files),
```

```
authorize {
        #
        #  The preprocess module takes care of sanitizing some bizarre
        #  attributes in the request, and turning them into attributes
        #  which are more standard.
        #

        preprocess

        #  This module takes care of EAP-MD5, EAP-TLS, and EAP-LEAP
        #  authentication.
        eap

        #  Look for IPASS style 'realm/', and if not found, look for
        #  '@realm', and decide whether or not to proxy, based on
        #  that.
        suffix

        #
        #  Read the 'users' file
        files
}


# Authentication.
#
#  This section lists which modules are available for authentication.
#  Note that it does NOT mean 'try each module in order'.  It means
#  that you have to have a module from the 'authorize' section add
#  a configuration attribute 'Auth-Type := FOO'.  That authentication type
#  is then used to pick the apropriate module from the list below.

authenticate {
        #
        #  Allow EAP authentication.
        eap
}

#
#  Pre-accounting.  Decide which accounting type to use.
#
preacct {
        preprocess

        #
        #  Look for IPASS-style 'realm/', and if not found, look for
```

```
        #  '@realm', and decide whether or not to proxy, based on
        #  that.
        #
        #  Accounting requests are generally proxied to the same
        #  home server as authentication requests.
        #       realmslash
        suffix

        #
        #  Read the 'acct_users' file
        files
}

#
#  Accounting.  Log the accounting data.
#
accounting {
        #
        #  Ensure that we have a semi-unique identifier for every
        #  request, and many NAS boxes are broken.
        acct_unique

        #
        #  Create a 'detail'ed log of the packets.
        detail
        #       daily

        unix            # wtmp file

        #
        #  For Simultaneous-Use tracking.

        radutmp
}


#  Session database, used for checking Simultaneous-Use. Either the radutmp
#  or rlm_sql module can handle this.
#  The rlm_sql module is *much* faster
session {
        radutmp
#       sql
}

#  Post-Authentication
#  Once we KNOW that the user has been authenticated, there are
```

```
#  additional steps we can take.
post-auth {
        #  Get an address from the IP Pool.
        #       main_pool
}

#
#  When the server decides to proxy a request to a home server,
#  the proxied request is first passed through the pre-proxy
#  stage
pre-proxy {
        #  If you want to have a log of packets proxied to a home
        #  server, un-comment the following line, and the
        #  'detail pre_proxy_log' section, above.
        #pre_proxy_log
}

#
#  When the server receives a reply to a request it proxied
#  to a home server, the request may be massaged here, in the
#  post-proxy stage.
#
post-proxy {
        #  reject the EAP request.
        #
        eap
}
```

C. CLIENTS CONFIGURATION FILE

```
#
# clients.conf - client configuration directives
#
# This file is included by default.  To disable it, you will need
# to modify the CLIENTS CONFIGURATION section of "radiusd.conf".
#
#########################################################################

#########################################################################
#
#  Definition of a RADIUS client (usually a NAS).
#
#  The information given here over rides anything given in the 'clients'
#  file, or in the 'naslist' file.  The configuration here contains
#  all of the information from those two files, and also allows for more
#  configuration items.
```

```
#
#  The "shortname" can be used for logging, and the "nastype",
#  "login" and "password" fields are mainly used for checkrad and are
#  optional.
#

#
#  Defines a RADIUS client.  The format is 'client [hostname|ip-address]'
#
#  '127.0.0.1' is another name for 'localhost'.  It is enabled by default,
#  to allow testing of the server after an initial installation.  If you
#  are not going to be permitting RADIUS queries from localhost, we suggest
#  that you delete, or comment out, this entry.
#
##############################Orhan ekledi#########################
#client 131.120.10.153 {
#secret = whatever
#shortname = CLIENT1
#}

client 131.120.8.145 {
secret = besiktas
shortname = AP1
}
#client 131.120.10.133 {
#secret = wahtever
#shortname = AP2
#}
########################################################################

client 127.0.0.1 {
        #
        #  The shared secret use to "encrypt" and "sign" packets between
        #  the NAS and FreeRADIUS.  You MUST change this secret from the
        #  default, otherwise it's not a secret any more!
        #
        #  The secret can be any string, up to 32 characters in length.
        #
        secret          = test

        #
        #  The short name is used as an alias for the fully qualified
        #  domain name, or the IP address.
        #
        shortname       = localhost
```

```
        #
        # the following three fields are optional, but may be used by
        # checkrad.pl for simultaneous use checks
        #

        #
        # The nastype tells 'checkrad.pl' which NAS-specific method to
        #  use to query the NAS for simultaneous use.
        #
        #  Permitted NAS types are:
        #
        #       cisco
        #       computone
        #       livingston
        #       max40xx
        #       multitech
        #       netserver
        #       pathras
        #       patton
        #       portslave
        #       tc
        #       usrhiper
        #       other           # for all other types

        #
        nastype     = other     # localhost isn't usually a NAS...

        #
        #  The following two configurations are for future use.
        #  The 'naspasswd' file is currently used to store the NAS
        #  login name and password, which is used by checkrad.pl
        #  when querying the NAS for simultaneous use.
        #
#       login       = !root
#       password    = someadminpas
}

#client some.host.org {
#       secret          = testing123
#       shortname       = localhost
#}

#
#  You can now specify one secret for a network of clients.
#  When a client request comes in, the BEST match is chosen.
#  i.e. The entry from the smallest possible network.
```

```
#
#client 192.168.0.0/24 {
#       secret          = testing123-1
#       shortname       = private-network-1
#}
#
#client 192.168.0.0/16 {
#       secret          = testing123-2
#       shortname       = private-network-2
#}


#client 10.10.10.10 {
#       # secret and password are mapped through the "secrets" file.
#       secret      = testing123
#       shortname   = liv1
#      # the following three fields are optional, but may be used by
#      # checkrad.pl for simultaneous usage checks
#       nastype     = livingston
#       login       = !root
#       password    = someadminpas
#}
```

D. USERS CONFIGURATION FILE

```
#
#       Please read the documentation file ../doc/processing_users_file,
#       or 'man 5 users' (after installing the server) for more information.
#
#       This file contains authentication security and configuration
#       information for each user.  Accounting requests are NOT processed
#       through this file.  Instead, see 'acct_users', in this directory.
#
#       The first field is the user's name and can be up to
#       253 characters in length.  This is followed (on the same line) with
#       the list of authentication requirements for that user.  This can
#       include password, comm server name, comm server port number, protocol
#       type (perhaps set by the "hints" file), and huntgroup name (set by
#       the "huntgroups" file).
#
#       If you are not sure why a particular reply is being sent by the
#       server, then run the server in debugging mode (radiusd -X), and
#       you will see which entries in this file are matched.
#
#       When an authentication request is received from the comm server,
#       these values are tested. Only the first match is used unless the
```

```
#       "Fall-Through" variable is set to "Yes".
#
#       A special user named "DEFAULT" matches on all usernames.
#       You can have several DEFAULT entries. All entries are processed
#       in the order they appear in this file. The first entry that
#       matches the login-request will stop processing unless you use
#       the Fall-Through variable.
#
#       If you use the database support to turn this file into a .db or .dbm
#       file, the DEFAULT entries _have_ to be at the end of this file and
#       you can't have multiple entries for one username.
#
#       You don't need to specify a password if you set Auth-Type += System
#       on the list of authentication requirements. The RADIUS server
#       will then check the system password file.
#
#       Indented (with the tab character) lines following the first
#       line indicate the configuration values to be passed back to
#       the comm server to allow the initiation of a user session.
#       This can include things like the PPP configuration values
#       or the host to log the user onto.
#
#       You can include another `users' file with `$INCLUDE users.other'
#


#
#       For a list of RADIUS attributes, and links to their definitions,
#       see:
#
#       http://www.freeradius.org/rfc/attributes.html
#


#
# Deny access for a specific user.  Note that this entry MUST
# be before any other 'Auth-Type' attribute which results in the user
# being authenticated.
#
# Note that there is NO 'Fall-Through' attribute, so the user will not
# be given any additional resources.
#
#lameuser       Auth-Type := Reject
#               Reply-Message = "Your account has been disabled."

#
# Deny access for a group of users.
#
```

```
# Note that there is NO 'Fall-Through' attribute, so the user will not
# be given any additional resources.
#
#DEFAULT    Group == "disabled", Auth-Type := Reject
#                   Reply-Message = "Your account has been disabled."
#

#
# This is a complete entry for "steve". Note that there is no Fall-Through
# entry so that no DEFAULT entry will be used, and the user will NOT
# get any attributes in addition to the ones listed here.
#
#steve  Auth-Type := Local, User-Password == "testing"
#         Service-Type = Framed-User,
#         Framed-Protocol = PPP,
#         Framed-IP-Address = 172.16.3.33,
#         Framed-IP-Netmask = 255.255.255.0,
#         Framed-Routing = Broadcast-Listen,
#         Framed-Filter-Id = "std.ppp",
#         Framed-MTU = 1500,
#         Framed-Compression = Van-Jacobsen-TCP-IP

#
# This is an entry for a user with a space in their name.
# Note the double quotes surrounding the name.
#
#"John Doe"    Auth-Type := Local, User-Password == "hello"
#                   Reply-Message = "Hello, %u"


#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#!!!!!!!!!!added by the designers!!!!!!!!!!!!!!!!!!!!!
#
newxpclient Auth-Type := EAP

test       Auth-Type := Local, User-Password == "test"
           Reply-Message = "hello,%u"
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#
# Dial user back and telnet to the default host for that port
#
#Deg    Auth-Type := Local, User-Password == "ge55ged"
#         Service-Type = Callback-Login-User,
#         Login-IP-Host = 0.0.0.0,
#         Callback-Number = "9,5551212",
#         Login-Service = Telnet,
```

```
#         Login-TCP-Port = Telnet

#
# Another complete entry. After the user "dialbk" has logged in, the
# connection will be broken and the user will be dialed back after which
# he will get a connection to the host "timeshare1".
#
#dialbkAuth-Type := Local, User-Password == "callme"
#         Service-Type = Callback-Login-User,
#         Login-IP-Host = timeshare1,
#         Login-Service = PortMaster,
#         Callback-Number = "9,1-800-555-1212"

#
# user "swilson" will only get a static IP number if he logs in with
# a framed protocol on a terminal server in Alphen (see the huntgroups file).
#
# Note that by setting "Fall-Through", other attributes will be added from
# the following DEFAULT entries
#
#swilson        Service-Type == Framed-User, Huntgroup-Name == "alphen"
#                Framed-IP-Address = 192.168.1.65,
#                Fall-Through = Yes

#
# If the user logs in as 'username.shell', then authenticate them
# against the system database, give them shell access, and stop processing
# the rest of the file.
#
#DEFAULT    Suffix == ".shell", Auth-Type := System
#                Service-Type = Login-User,
#                Login-Service = Telnet,
#                Login-IP-Host = your.shell.machine


#
# The rest of this file contains the several DEFAULT entries.
# DEFAULT entries match with all login names.
# Note that DEFAULT entries can also Fall-Through (see first entry).
# A name-value pair from a DEFAULT entry will _NEVER_ override
# an already existing name-value pair.
#

#
# First setup all accounts to be checked against the UNIX /etc/passwd.
# (Unless a password was already given earlier in this file).
```

```
#
DEFAULT      Auth-Type := System
        Fall-Through = 1


#
# Set up different IP address pools for the terminal servers.
# Note that the "+" behind the IP address means that this is the "base"
# IP address. The Port-Id (S0, S1 etc) will be added to it.
#
#DEFAULT    Service-Type == Framed-User, Huntgroup-Name == "alphen"
#                Framed-IP-Address = 192.168.1.32+,
#                Fall-Through = Yes

#DEFAULT    Service-Type == Framed-User, Huntgroup-Name == "delft"
#                Framed-IP-Address = 192.168.2.32+,
#                Fall-Through = Yes

#
# Defaults for all framed connections.
#
DEFAULT      Service-Type == Framed-User
        Framed-IP-Address = 255.255.255.254,
        Framed-MTU = 576,
        Service-Type = Framed-User,
        Fall-Through = Yes

#
# Default for PPP: dynamic IP address, PPP mode, VJ-compression.
# NOTE: we do not use Hint = "PPP", since PPP might also be auto-detected
#        by the terminal server in which case there may not be a "P" suffix.
#        The terminal server sends "Framed-Protocol = PPP" for auto PPP.
#
DEFAULT      Framed-Protocol == PPP
        Framed-Protocol = PPP,
        Framed-Compression = Van-Jacobson-TCP-IP

#
# Default for CSLIP: dynamic IP address, SLIP mode, VJ-compression.
#
DEFAULT      Hint == "CSLIP"
        Framed-Protocol = SLIP,
        Framed-Compression = Van-Jacobson-TCP-IP

#
# Default for SLIP: dynamic IP address, SLIP mode.
```

```
#
DEFAULT      Hint == "SLIP"
        Framed-Protocol = SLIP

#
# Last default: rlogin to our main server.
#
#DEFAULT
#       Service-Type = Login-User,
#       Login-Service = Rlogin,
#       Login-IP-Host = shellbox.ispdomain.com

# #
# # Last default: shell on the local terminal server.
# #
# DEFAULT
#       Service-Type = Shell-User

# On no match, the user is denied access.
```

E. RADIUSD RUNNING SCRIPT

```
#!/bin/sh -x

LD_LIBRARY_PATH=/usr/local/openssl/lib
LD_PRELOAD=/usr/local/openssl/lib/libcrypto.so

export LD_LIBRARY_PATH LD_PRELOAD

/usr/local/sbin/radiusd -X -A $@
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E

A. AUTHENTICATION SERVER SUCCESSFUL AUTHENTICATION LOGS

```
Starting - reading configuration files ...
reread_config:  reading radiusd.conf
Config:   including file: /etc/raddb/proxy.conf
Config:   including file: /etc/raddb/clients.conf
Config:   including file: /etc/raddb/snmp.conf
Config:   including file: /etc/raddb/sql.conf
 main: prefix = "/usr/local"
 main: localstatedir = "/usr/local/var"
 main: logdir = "/usr/local/var/log/radius"
 main: libdir = "/usr/local/lib"
 main: radacctdir = "/usr/local/var/log/radius/radacct"
 main: hostname_lookups = no
 main: max_request_time = 30
 main: cleanup_delay = 5
 main: max_requests = 1024
 main: delete_blocked_requests = 0
 main: port = 0
 main: allow_core_dumps = no
 main: log_stripped_names = no
 main: log_file = "/usr/local/var/log/radius/radius.log"
 main: log_auth = no
 main: log_auth_badpass = no
 main: log_auth_goodpass = no
 main: pidfile = "/usr/local/var/run/radiusd/radiusd.pid"
 main: user = "(null)"
 main: group = "(null)"
 main: usercollide = no
 main: lower_user = "no"
 main: lower_pass = "no"
 main: nospace_user = "no"
 main: nospace_pass = "no"
 main: checkrad = "/usr/local/sbin/checkrad"
 main: proxy_requests = yes
 proxy: retry_delay = 5
 proxy: retry_count = 3
 proxy: synchronous = no
 proxy: default_fallback = yes
 proxy: dead_time = 120
 proxy: post_proxy_authorize = yes
 proxy: wake_all_if_all_dead = no
 security: max_attributes = 200
 security: reject_delay = 1
 security: status_server = no
 main: debug_level = 0
read_config_files:  reading dictionary
read_config_files:  reading naslist
Using deprecated naslist file.  Support for this will go away soon.
read_config_files:  reading clients
Using deprecated clients file.  Support for this will go away soon.
read_config_files:  reading realms
```

```
Using deprecated realms file.  Support for this will go away soon.
radiusd:  entering modules setup
Module: Library search path is /usr/local/lib
Module: Loaded expr
Module: Instantiated expr (expr)
Module: Loaded System
 unix: cache = yes
 unix: passwd = "/etc/passwd"
 unix: shadow = "/etc/shadow"
 unix: group = "/etc/group"
 unix: radwtmp = "/usr/local/var/log/radius/radwtmp"
 unix: usegroup = no
 unix: cache_reload = 600
HASH:  Reinitializing hash structures and lists for caching...
  HASH:  user root found in hashtable bucket 11726
  HASH:  user bin found in hashtable bucket 86651
  HASH:  user daemon found in hashtable bucket 11668
  HASH:  user adm found in hashtable bucket 26466
  HASH:  user lp found in hashtable bucket 54068
  HASH:  user sync found in hashtable bucket 42895
  HASH:  user shutdown found in hashtable bucket 71746
  HASH:  user halt found in hashtable bucket 7481
  HASH:  user mail found in hashtable bucket 79471
  HASH:  user news found in hashtable bucket 5375
  HASH:  user uucp found in hashtable bucket 38541
  HASH:  user operator found in hashtable bucket 21748
  HASH:  user games found in hashtable bucket 47657
  HASH:  user gopher found in hashtable bucket 47357
  HASH:  user ftp found in hashtable bucket 56226
  HASH:  user nobody found in hashtable bucket 99723
  HASH:  user rpm found in hashtable bucket 72383
  HASH:  user vcsa found in hashtable bucket 25959
  HASH:  user nscd found in hashtable bucket 36306
  HASH:  user sshd found in hashtable bucket 71560
  HASH:  user rpc found in hashtable bucket 72373
  HASH:  user rpcuser found in hashtable bucket 552
  HASH:  user nfsnobody found in hashtable bucket 51830
  HASH:  user mailnull found in hashtable bucket 78086
  HASH:  user smmsp found in hashtable bucket 13600
  HASH:  user pcap found in hashtable bucket 55326
  HASH:  user xfs found in hashtable bucket 17213
  HASH:  user ntp found in hashtable bucket 21418
  HASH:  user gdm found in hashtable bucket 50360
  HASH:  user oozan found in hashtable bucket 94479
  HASH:  user amanda found in hashtable bucket 72438
HASH:  Stored 31 entries from /etc/passwd
HASH:  Stored 39 entries from /etc/group
Module: Instantiated unix (unix)
Module: Loaded eap
 eap: default_eap_type = "tls"
 eap: timer_expire = 60
 eap: ignore_unknown_eap_types = no
 tls: rsa_key_exchange = no
 tls: dh_key_exchange = yes
 tls: rsa_key_length = 512
 tls: dh_key_length = 512
 tls: verify_depth = 0
```

```
 tls: CA_path = "(null)"
 tls: pem_file_type = yes
 tls: private_key_file = "/etc/1x/newradius.pem"
 tls: certificate_file = "/etc/1x/newradius.pem"
 tls: CA_file = "/etc/1x/root.pem"
 tls: private_key_password = "whatever"
 tls: dh_file = "/etc/1x/DH"
 tls: random_file = "/etc/1x/random"
 tls: fragment_size = 1024
 tls: include_length = yes
 tls: check_crl = no
rlm_eap: Loaded and initialized type tls
rlm_eap: Loaded and initialized type mschapv2
Module: Instantiated eap (eap)
Module: Loaded preprocess
 preprocess: huntgroups = "/etc/raddb/huntgroups"
 preprocess: hints = "/etc/raddb/hints"
 preprocess: with_ascend_hack = no
 preprocess: ascend_channels_per_line = 23
 preprocess: with_ntdomain_hack = no
 preprocess: with_specialix_jetstream_hack = no
 preprocess: with_cisco_vsa_hack = no
Module: Instantiated preprocess (preprocess)
Module: Loaded realm
 realm: format = "suffix"
 realm: delimiter = "@"
Module: Instantiated realm (suffix)
Module: Loaded files
 files: usersfile = "/etc/raddb/users"
 files: acctusersfile = "/etc/raddb/acct_users"
 files: preproxy_usersfile = "/etc/raddb/preproxy_users"
 files: compat = "no"
Module: Instantiated files (files)
Module: Loaded Acct-Unique-Session-Id
 acct_unique: key = "User-Name, Acct-Session-Id, NAS-IP-Address,
Client-IP-Address, NAS-Port-Id"
Module: Instantiated acct_unique (acct_unique)
Module: Loaded detail
 detail: detailfile = "/usr/local/var/log/radius/radacct/%{Client-IP-
Address}/detail-%Y%m%d"
 detail: detailperm = 384
 detail: dirperm = 493
 detail: locking = no
Module: Instantiated detail (detail)
Module: Loaded radutmp
 radutmp: filename = "/usr/local/var/log/radius/radutmp"
 radutmp: username = "%{User-Name}"
 radutmp: case_sensitive = yes
 radutmp: check_with_nas = yes
 radutmp: perm = 384
 radutmp: callerid = yes
Module: Instantiated radutmp (radutmp)
Listening on IP address *, ports 1812/udp and 1813/udp, with proxy on
1814/udp.
Ready to process requests.
rad_recv: Access-Request packet from host 131.120.8.145:32804, id=0,
length=160
```

```
      User-Name = "newxpclient"
      NAS-IP-Address = 131.120.8.145
      NAS-Port = 1
      Called-Station-Id = "00-05-5D-D9-8D-AE:test"
      Calling-Station-Id = "00-05-5D-D9-57-59"
      Framed-MTU = 2304
      NAS-Port-Type = Wireless-802.11
      Connect-Info = "CONNECT 11Mbps 802.11b"
      EAP-Message = "\002\001\000\020\001newxpclient"
      Message-Authenticator = 0x7b47883e05d44aa13d69442f35d1178f
modcall: entering group authorize for request 0
  modcall[authorize]: module "preprocess" returns ok for request 0
  rlm_eap: EAP packet type response id 1 length 16
  rlm_eap: No EAP Start, assuming it's an on-going EAP conversation
  modcall[authorize]: module "eap" returns updated for request 0
    rlm_realm: No '@' in User-Name = "newxpclient", looking up realm
NULL
    rlm_realm: No such realm "NULL"
  modcall[authorize]: module "suffix" returns noop for request 0
    users: Matched newxpclient at 101
  modcall[authorize]: module "files" returns ok for request 0
modcall: group authorize returns updated for request 0
  rad_check_password:  Found Auth-Type EAP
auth: type "EAP"
modcall: entering group authenticate for request 0
  rlm_eap: EAP Identity
  rlm_eap: processing type tls
 rlm_eap_tls: Requiring client certificate
  rlm_eap_tls: Initiate
  rlm_eap_tls: Start returned 1
  modcall[authenticate]: module "eap" returns handled for request 0
modcall: group authenticate returns handled for request 0
Sending Access-Challenge of id 0 to 131.120.8.145:32804
      EAP-Message = "\001\002\000\006\r "
      Message-Authenticator = 0x00000000000000000000000000000000
      State = 0xcabe64f58e2e52c0326344aeaf7ff16d
Finished request 0
……
……
……
Going to the next request
Waking up in 1 seconds...
rad_recv: Access-Request packet from host 131.120.8.145:32804, id=12,
length=1164
      User-Name = "newxpclient"
      NAS-IP-Address = 131.120.8.145
      NAS-Port = 1
      Called-Station-Id = "00-05-5D-D9-8D-AE:test"
      Calling-Station-Id = "00-05-5D-D9-57-59"
      Framed-MTU = 2304
      NAS-Port-Type = Wireless-802.11
      Connect-Info = "CONNECT 11Mbps 802.11b"
      EAP-Message =
"\002\017\003\344\r\200\000\000\003\332\026\003\001\003\252\013\000\002
\232\000\002\227\000\002\2240\202\002\2200\202\001\371\240\003\002\001\
002\002\001\0020\r\006\t*\206H\206\367\r\001\001\004\005\0000v1\0130\t\
006\003U\004\006\023\002US1\0230\021\006\003U\004\010\023\nCalifornia1\
```

```
0210\017\006\003U\004\007\023\010Monterey1\r0\013\006\003U\004\n\023\00
4NPGS1\r0\013\006\003U\004\013\023\004SAAM1!0\037\006\t*\206H\206\367\r
\001\t\001\026\022oozan@nps.navy.mil0\036\027\r040115004027Z\027\r05011
4004027Z0\201\214"
      EAP-Message =
"\007\023\010Monterey1\r0\013\006\003U\004\n\023\004NPGS1\r0\013\006\00
3U\004\013\023\004SAAM1\0240\022\006\003U\004\003\023\013newxpclient1!0
\037\006\t*\206H\206\367\r\001\t\001\026\022oozan@nps.navy.mil0\201\237
0\r\006\t*\206H\206\367\r\001\001\001\005\000\003\201\215\0000\201\211\
002\201\201\000\266+\244#B\341Y\335\215\314\272\322(P\263\353\355N\311\
037\235\323\203V4\256\275\311\277=\337I2|\n\312\026\225
\006\335w\023\014\265\302\350\276\335\330\234\224\346\373m\226\027\001\
n\002Y\322 ?y]\352\026\231%iF"
      EAP-Message =
"f\277\002\003\001\000\001\243\0270\0250\023\006\003U\035%\004\0140\n\0
06\010+\006\001\005\005\007\003\0020\r\006\t*\206H\206\367\r\001\001\00
4\005\000\003\201\201\000\301\211\227=\321\366c-
\357Z\022:9\231\016\216A\363\352\356\276\246E\341\364+X"\352\316]\010\0
31\335\251=Ul\207D\003\000\330\312\361\002\247\210\0143
o\327\276l,\0242\306\307\267\311:/\373N\321\227\355\000\377\237x\302u\2
70DBr\006|\027SF\003\003\240\341\364X\367\203\277\315\302\027\331\331\2
16xp \245j@\271\2244\376:\364\330\374>\237>\367M\301"
      EAP-Message =
"\003=x7*\t*Aj\364\017\304\336\251\321\345\003\036\371y\252\253Va\211\3
76\202\240\033P\222\210"\000\374Er\030U\204\326\355W\217n9\370B[\315Y]U
\244h\375\r\332\033/\017\000\000\202\000\200#Di[\230g\337n\266NRA\240QN
/c\261$\013\323v\344S\326T\2370RM\n\331\255\343}A\315\332\265\242\220\2
20~\367\335\221Vd\227\266&z3EY\306\336x\206j\333\235\030\033\274\350gO7
\036\254\336\311\037\\0\213\276L\322\224U\321\302Z\001@L\221\261\301'\0
35^\2769@\270G\351M!\217V>\364\3557\246}\235\350\300\3366Y-
7u\033\037\250q\331o\223\340"
      State = 0x82f684dced5c2fa94166b1538025a5b1
      Message-Authenticator = 0xece082dddb31cdc0a636c05351531ab4
modcall: entering group authorize for request 12
  modcall[authorize]: module "preprocess" returns ok for request 12
  rlm_eap: EAP packet type response id 15 length 253
  rlm_eap: No EAP Start, assuming it's an on-going EAP conversation
  modcall[authorize]: module "eap" returns updated for request 12
    rlm_realm: No '@' in User-Name = "newxpclient", looking up realm
NULL
    rlm_realm: No such realm "NULL"
  modcall[authorize]: module "suffix" returns noop for request 12
    users: Matched newxpclient at 101
  modcall[authorize]: module "files" returns ok for request 12
modcall: group authorize returns updated for request 12
  rad_check_password:  Found Auth-Type EAP
auth: type "EAP"
modcall: entering group authenticate for request 12
  rlm_eap: Request found, released from the list
  rlm_eap: EAP_TYPE - tls
  rlm_eap: processing type tls
  rlm_eap_tls: Authenticate
  rlm_eap_tls: processing TLS
rlm_eap_tls:  Length Included
  eaptls_verify returned 11
  rlm_eap_tls: <<< TLS 1.0 Handshake [length 029e], Certificate
chain-depth=1,
```

```
error=0
--> User-Name = newxpclient
--> BUF-Name = ? ??????
--> subject =
/C=US/ST=California/L=Monterey/O=NPGS/OU=SAAM/Email=oozan@nps.navy.mil
--> issuer  =
/C=US/ST=California/L=Monterey/O=NPGS/OU=SAAM/Email=oozan@nps.navy.mil
--> verify return:1
chain-depth=0,
error=0
--> User-Name = newxpclient
--> BUF-Name = newxpclient
--> subject =
/C=US/ST=California/L=Monterey/O=NPGS/OU=SAAM/CN=newxpclient/Email=ooza
n@nps.navy.mil
--> issuer  =
/C=US/ST=California/L=Monterey/O=NPGS/OU=SAAM/Email=oozan@nps.navy.mil
--> verify return:1
TLS_accept: SSLv3 read client certificate A
  rlm_eap_tls: <<< TLS 1.0 Handshake [length 0086], ClientKeyExchange
TLS_accept: SSLv3 read client key exchange A
  rlm_eap_tls: <<< TLS 1.0 Handshake [length 0086], CertificateVerify
TLS_accept: SSLv3 read certificate verify A
  rlm_eap_tls: <<< TLS 1.0 ChangeCipherSpec [length 0001]
  rlm_eap_tls: <<< TLS 1.0 Handshake [length 0010], Finished
TLS_accept: SSLv3 read finished A
  rlm_eap_tls: >>> TLS 1.0 ChangeCipherSpec [length 0001]
TLS_accept: SSLv3 write change cipher spec A
  rlm_eap_tls: >>> TLS 1.0 Handshake [length 0010], Finished
TLS_accept: SSLv3 write finished A
TLS_accept: SSLv3 flush data
undefined: SSL negotiation finished successfully
SSL Connection Established
  eaptls_process returned 13
  modcall[authenticate]: module "eap" returns handled for request 12
modcall: group authenticate returns handled for request 12
Sending Access-Challenge of id 12 to 131.120.8.145:32804
      EAP-Message =
"\001\020\0005\r\200\000\000\000+\024\003\001\000\001\001\026\003\001\0
00
\344\244\001\311\376T\242\022K|[2\365\345\262~2\274gFKR\274\334\271\003
\247\215\037\321q("
      Message-Authenticator = 0x00000000000000000000000000000000
      State = 0x2c02871aafebad85e9d5602736783471
Finished request 12
Going to the next request
Waking up in 1 seconds...
rad_recv: Access-Request packet from host 131.120.8.145:32804, id=13,
length=168
      User-Name = "newxpclient"
      NAS-IP-Address = 131.120.8.145
      NAS-Port = 1
      Called-Station-Id = "00-05-5D-D9-8D-AE:test"
      Calling-Station-Id = "00-05-5D-D9-57-59"
      Framed-MTU = 2304
      NAS-Port-Type = Wireless-802.11
      Connect-Info = "CONNECT 11Mbps 802.11b"
```

```
      EAP-Message = "\002\020\000\006\r"
      State = 0x2c02871aafebad85e9d5602736783471
      Message-Authenticator = 0xb1f4cfdc73e426a656047670cc908ef6
modcall: entering group authorize for request 13
  modcall[authorize]: module "preprocess" returns ok for request 13
  rlm_eap: EAP packet type response id 16 length 6
  rlm_eap: No EAP Start, assuming it's an on-going EAP conversation
  modcall[authorize]: module "eap" returns updated for request 13
    rlm_realm: No '@' in User-Name = "newxpclient", looking up realm
NULL
    rlm_realm: No such realm "NULL"
  modcall[authorize]: module "suffix" returns noop for request 13
    users: Matched newxpclient at 101
  modcall[authorize]: module "files" returns ok for request 13
modcall: group authorize returns updated for request 13
  rad_check_password:  Found Auth-Type EAP
auth: type "EAP"
modcall: entering group authenticate for request 13
  rlm_eap: Request found, released from the list
  rlm_eap: EAP_TYPE - tls
  rlm_eap: processing type tls
  rlm_eap_tls: Authenticate
  rlm_eap_tls: processing TLS
rlm_eap_tls: Received EAP-TLS ACK message
  rlm_eap_tls: ack handshake is finished
  eaptls_verify returned 3
  eaptls_process returned 3
  rlm_eap: Freeing handler
  modcall[authenticate]: module "eap" returns ok for request 13
modcall: group authenticate returns ok for request 13
Sending Access-Accept of id 13 to 131.120.8.145:32804
      MS-MPPE-Recv-Key =
0xe9545b180975cdfb5d0f982189e03b43602f6c475e4ee66d7d24783c056f314c
      MS-MPPE-Send-Key =
0x1f44ce961ecf533c728e731dff144b918ed1a420fd83185e4ecc6b3c686a08b9
      EAP-Message = "\003\020\000\004"
      Message-Authenticator = 0x00000000000000000000000000000000
      User-Name = "newxpclient"
Finished request 13
Going to the next request
Waking up in 1 seconds...
--- Walking the entire request list ---
Waking up in 2 seconds...
--- Walking the entire request list ---
Cleaning up request 4 ID 4 with timestamp 4006d26e
Cleaning up request 5 ID 5 with timestamp 4006d26e
Cleaning up request 6 ID 6 with timestamp 4006d26e
Cleaning up request 7 ID 7 with timestamp 4006d26e
Cleaning up request 8 ID 8 with timestamp 4006d26e
Waking up in 3 seconds...
--- Walking the entire request list ---
Cleaning up request 9 ID 9 with timestamp 4006d271
Cleaning up request 10 ID 10 with timestamp 4006d271
Cleaning up request 11 ID 11 with timestamp 4006d271
Cleaning up request 12 ID 12 with timestamp 4006d271
Cleaning up request 13 ID 13 with timestamp 4006d271
Nothing to do.  Sleeping until we see a request.
```

B. AUTHENTICATOR SUCCESSFUL SUPPLICANT AUTHENTICATION LOG

```
Opening raw packet socket for ifindex 4
Using interface wlan0ap with hwaddr 00:05:5d:d9:8d:ae and ssid 'test'
Default WEP key - hexdump(len=5): 8a 0a ef db b7
Flushing old station entries
Deauthenticate all stations
Received 30 bytes management frame
  dump: b0 00 02 01 00 05 5d d9 8d ae 00 05 5d d9 57 59 00 05 5d d9 8d
ae a0 01 00 00 01 00 00 00
MGMT
mgmt::auth
authentication: STA=00:05:5d:d9:57:59 auth_alg=0 auth_transaction=1
status_code=0
  New STA
Station 00:05:5d:d9:57:59 authentication OK (open system)
Received 30 bytes management frame
  dump: b2 00 3a 01 00 05 5d d9 57 59 00 05 5d d9 8d ae 00 05 5d d9 8d
ae 20 44 00 00 02 00 00 00
MGMT (TX callback) ACK
mgmt::auth cb
Station 00:05:5d:d9:57:59 authenticated
Received 40 bytes management frame
  dump: 00 00 02 01 00 05 5d d9 8d ae 00 05 5d d9 57 59 00 05 5d d9 8d
ae b0 01 01 00 01 00 00 04 74 65 73 74 01 04 82 84 0b 16
MGMT
mgmt::assoc_req
association request: STA=00:05:5d:d9:57:59 capab_info=0x01
listen_interval=1
  new AID 1
Station 00:05:5d:d9:57:59 association OK (aid 1)
Received 36 bytes management frame
  dump: 12 00 3a 01 00 05 5d d9 57 59 00 05 5d d9 8d ae 00 05 5d d9 8d
ae 30 44 01 00 00 00 01 c0 01 04 82 84 0b 16
MGMT (TX callback) ACK
mgmt::assoc_resp cb
Station 00:05:5d:d9:57:59 associated (aid 1)
IEEE 802.1X: Start authentication for new station 00:05:5d:d9:57:59
IEEE 802.1X: 00:05:5d:d9:57:59 AUTH_PAE entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 AUTH_KEY_TX entering state
NO_KEY_TRANSMIT
IEEE 802.1X: 00:05:5d:d9:57:59 AUTH_PAE entering state DISCONNECTED
IEEE 802.1X: Unauthorizing station 00:05:5d:d9:57:59
IEEE 802.1X: Sending canned EAP packet FAILURE to 00:05:5d:d9:57:59
(identifier 0)
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state IDLE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 AUTH_PAE entering state CONNECTING
IEEE 802.1X: Sending EAP Request-Identity to 00:05:5d:d9:57:59
(identifier 1)
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
```

```
Received 40 bytes management frame
  dump: 0a 02 3a 01 00 05 5d d9 57 59 00 05 5d d9 8d ae 00 05 5d d9 8d
ae 40 44 aa aa 03 00 00 00 88 8e 01 00 00 04 04 00 00 04
DATA (TX callback) ACK
Received 46 bytes management frame
  dump: 0a 02 3a 01 00 05 5d d9 57 59 00 05 5d d9 8d ae 00 05 5d d9 8d
ae 50 44 aa aa 03 00 00 00 88 8e 01 00 00 0a 01 01 00 0a 01 68 65 6c 6c
6f
DATA (TX callback) ACK
Received 37 bytes management frame
  dump: 08 01 02 01 00 05 5d d9 8d ae 00 05 5d d9 57 59 00 05 5d d9 8d
ae c0 01 aa aa 03 00 00 00 88 8e 01 01 00 00 00
DATA
IEEE 802.1X: 5 bytes from 00:05:5d:d9:57:59
   IEEE 802.1X: version=1 type=1 length=0
   ignoring 1 extra octets after IEEE 802.1X packet
   EAPOL-Start
IEEE 802.1X: 00:05:5d:d9:57:59 AUTH_PAE entering state CONNECTING
IEEE 802.1X: Sending EAP Request-Identity to 00:05:5d:d9:57:59
(identifier 1)
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
Received 46 bytes management frame
  dump: 0a 02 3a 01 00 05 5d d9 57 59 00 05 5d d9 8d ae 00 05 5d d9 8d
ae 60 44 aa aa 03 00 00 00 88 8e 01 00 00 0a 01 01 00 0a 01 68 65 6c 6c
6f
DATA (TX callback) ACK
Received 52 bytes management frame
  dump: 08 01 02 01 00 05 5d d9 8d ae 00 05 5d d9 57 59 00 05 5d d9 8d
ae d0 01 aa aa 03 00 00 00 88 8e 01 00 00 10 02 01 00 10 01 6e 65 77 78
70 63 6c 69 65 6e 74
DATA
IEEE 802.1X: 20 bytes from 00:05:5d:d9:57:59
   IEEE 802.1X: version=1 type=0 length=16
   EAP: code=2 identifier=1 length=16 (response)
   EAP Response-Identity
IEEE 802.1X: 00:05:5d:d9:57:59 AUTH_PAE entering state AUTHENTICATING
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state RESPONSE
Encapsulating EAP message into a RADIUS packet
Sending RADIUS message to authentication server
RADIUS message: code=1 (Access-Request) identifier=0 length=160
   Attribute 1 (User-Name) length=13
      Value: 'newxpclient'
   Attribute 4 (NAS-IP-Address) length=6
      Value: 131.120.8.145
   Attribute 5 (NAS-Port) length=6
      Value: 1
   Attribute 30 (Called-Station-Id) length=24
      Value: '00-05-5D-D9-8D-AE:test'
   Attribute 31 (Calling-Station-Id) length=19
      Value: '00-05-5D-D9-57-59'
   Attribute 12 (Framed-MTU) length=6
      Value: 2304
   Attribute 61 (NAS-Port-Type) length=6
      Value: 19
   Attribute 77 (Connect-Info) length=24
      Value: 'CONNECT 11Mbps 802.11b'
```

```
   Attribute 79 (EAP-Message) length=18
   Attribute 80 (Message-Authenticator) length=18
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
Received 52 bytes management frame
  dump: 08 01 02 01 00 05 5d d9 8d ae 00 05 5d d9 57 59 00 05 5d d9 8d
ae e0 01 aa aa 03 00 00 00 88 8e 01 00 00 10 02 01 00 10 01 6e 65 77 78
70 63 6c 69 65 6e 74
DATA
IEEE 802.1X: 20 bytes from 00:05:5d:d9:57:59
   IEEE 802.1X: version=1 type=0 length=16
   EAP: code=2 identifier=1 length=16 (response)
   EAP Response-Identity
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
Received 64 bytes from authentication server
Received RADIUS message
RADIUS message: code=11 (Access-Challenge) identifier=0 length=64
   Attribute 79 (EAP-Message) length=8
   Attribute 80 (Message-Authenticator) length=18
   Attribute 24 (State) length=18
RADIUS packet matching with station 00:05:5d:d9:57:59
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state REQUEST
IEEE 802.1X: Sending EAP Packet to 00:05:5d:d9:57:59 (identifier 2)
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
Received 42 bytes management frame
  dump: 0a 02 3a 01 00 05 5d d9 57 59 00 05 5d d9 8d ae 00 05 5d d9 8d
ae 90 44 aa aa 03 00 00 00 88 8e 01 00 00 06 01 02 00 06 0d 20
DATA (TX callback) ACK
Received 148 bytes management frame
  dump: 08 01 02 01 00 05 5d d9 8d ae 00 05 5d d9 57 59 00 05 5d d9 8d
ae f0 01 aa aa 03 00 00 00 88 8e 01 00 00 70 02 02 00 70 0d 80 00 00 00
66 16 03 01 00 61 01 00 00 5d 03 01 40 06 d1 58 59 2c af d8 f8 1e 81 19
ca 6e c5 66 34 d6 a6 28 85 47 61 eb 69 e8 c9 3c 8f a4 a0 00 20 81 90 a4
11 52 ad 3b 0b 8f f1 cd 8a 98 ce 08 51 41 c8 f5 75 34 35 54 84 9b 7a 08
f5 73 5d d0 82 00 16 00 04 00 05 00 0a 00 09 00 64 00 62 00 03 00 06 00
13 00 12 00 63 01 00
DATA
IEEE 802.1X: 116 bytes from 00:05:5d:d9:57:59
   IEEE 802.1X: version=1 type=0 length=112
   EAP: code=2 identifier=2 length=112 (response)
   EAP Response-TLS
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state RESPONSE
Encapsulating EAP message into a RADIUS packet
Sending RADIUS message to authentication server
……
……
……

RADIUS message: code=1 (Access-Request) identifier=7 length=1164
   Attribute 1 (User-Name) length=13
      Value: 'newxpclient'
   Attribute 4 (NAS-IP-Address) length=6
      Value: 131.120.8.145
   Attribute 5 (NAS-Port) length=6
      Value: 1
   Attribute 30 (Called-Station-Id) length=24
```

```
      Value: '00-05-5D-D9-8D-AE:test'
   Attribute 31 (Calling-Station-Id) length=19
      Value: '00-05-5D-D9-57-59'
   Attribute 12 (Framed-MTU) length=6
      Value: 2304
   Attribute 61 (NAS-Port-Type) length=6
      Value: 19
   Attribute 77 (Connect-Info) length=24
      Value: 'CONNECT 11Mbps 802.11b'
   Attribute 79 (EAP-Message) length=255
   Attribute 79 (EAP-Message) length=255
   Attribute 79 (EAP-Message) length=255
   Attribute 79 (EAP-Message) length=239
   Attribute 24 (State) length=18
   Attribute 80 (Message-Authenticator) length=18
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
Received 111 bytes from authentication server
Received RADIUS message
RADIUS message: code=11 (Access-Challenge) identifier=7 length=111
   Attribute 79 (EAP-Message) length=55
   Attribute 80 (Message-Authenticator) length=18
   Attribute 24 (State) length=18
RADIUS packet matching with station 00:05:5d:d9:57:59
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state REQUEST
IEEE 802.1X: Sending EAP Packet to 00:05:5d:d9:57:59 (identifier 11)
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
Received 89 bytes management frame
  dump: 0a 02 3a 01 00 05 5d d9 57 59 00 05 5d d9 8d ae 00 05 5d d9 8d
ae 90 47 aa aa 03 00 00 00 88 8e 01 00 00 35 01 0b 00 35 0d 80 00 00 00
2b 14 03 01 00 01 01 16 03 01 00 20 00 49 d1 0f a2 35 ca 70 91 52 96 46
fc cc 25 cc 65 ea 81 31 7e 21 5f 2a f0 0c df 05 06 37 55 5b
DATA (TX callback) ACK
Received 42 bytes management frame
  dump: 08 01 02 01 00 05 5d d9 8d ae 00 05 5d d9 57 59 00 05 5d d9 8d
ae 60 03 aa aa 03 00 00 00 88 8e 01 00 00 06 02 0b 00 06 0d 00
DATA
IEEE 802.1X: 10 bytes from 00:05:5d:d9:57:59
   IEEE 802.1X: version=1 type=0 length=6
   EAP: code=2 identifier=11 length=6 (response)
   EAP Response-TLS
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state RESPONSE
Encapsulating EAP message into a RADIUS packet
Sending RADIUS message to authentication server
RADIUS message: code=1 (Access-Request) identifier=8 length=168
   Attribute 1 (User-Name) length=13
      Value: 'newxpclient'
   Attribute 4 (NAS-IP-Address) length=6
      Value: 131.120.8.145
   Attribute 5 (NAS-Port) length=6
      Value: 1
   Attribute 30 (Called-Station-Id) length=24
      Value: '00-05-5D-D9-8D-AE:test'
   Attribute 31 (Calling-Station-Id) length=19
      Value: '00-05-5D-D9-57-59'
   Attribute 12 (Framed-MTU) length=6
```

```
      Value: 2304
   Attribute 61 (NAS-Port-Type) length=6
      Value: 19
   Attribute 77 (Connect-Info) length=24
      Value: 'CONNECT 11Mbps 802.11b'
   Attribute 79 (EAP-Message) length=8
   Attribute 24 (State) length=18
   Attribute 80 (Message-Authenticator) length=18
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
Received 173 bytes from authentication server
Received RADIUS message
RADIUS message: code=2 (Access-Accept) identifier=8 length=173
   Attribute 26 (Vendor-Specific) length=58
   Attribute 26 (Vendor-Specific) length=58
   Attribute 79 (EAP-Message) length=6
   Attribute 80 (Message-Authenticator) length=18
   Attribute 1 (User-Name) length=13
      Value: 'newxpclient'
RADIUS packet matching with station 00:05:5d:d9:57:59
MS-MPPE-Send-Key (len=32): 46 2b a4 7c 88 4a d5 f8 f4 54 f1 ba fd df 56
19 a4 23 f8 aa 5b e4 f3 5d 5a 19 a7 94 ac 7a d4 d3
MS-MPPE-Recv-Key (len=32): 0d 74 4b e2 30 ac 23 ed 3b f1 e1 ba 66 55 c7
a4 b5 07 78 7a c5 91 43 04 46 4c ce 82 60 ab 2a 57
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state SUCCESS
IEEE 802.1X: Sending EAP Packet to 00:05:5d:d9:57:59 (identifier 11)
IEEE 802.1X: 00:05:5d:d9:57:59 REAUTH_TIMER entering state INITIALIZE
IEEE 802.1X: 00:05:5d:d9:57:59 AUTH_KEY_TX entering state KEY_TRANSMIT
IEEE 802.1X: Sending EAPOL-Key(s) to 00:05:5d:d9:57:59 (identifier 11)
IEEE 802.1X: Sending EAPOL-Key to 00:05:5d:d9:57:59 (broadcast index=1)
Individual WEP key - hexdump(len=5): f8 bf 5d 05 d2
IEEE 802.1X: Sending EAPOL-Key to 00:05:5d:d9:57:59 (unicast index=0)
IEEE 802.1X: 00:05:5d:d9:57:59 AUTH_PAE entering state AUTHENTICATED
IEEE 802.1X: Authorizing station 00:05:5d:d9:57:59
IEEE 802.1X: 00:05:5d:d9:57:59 BE_AUTH entering state IDLE

Signal 2 received - terminating
Removing station 00:05:5d:d9:57:59
Flushing old station entries
Deauthenticate all stations
```

LIST OF REFERENCES

1. Tom Karygiannis and Les Owens, *Wireless Network Security*, National Institute for Standarts and Technology, Special Publication 800-48.

2. James F. Kurose and Keith W. Ross, *Computer Networking,* Addison Wesley Longman, 2001.

3. Institute of Electrical and Electronics Engineers, *IEEE Standard for Local and Metropolitan Area Networks Port-Based Network Access Control,* IEEE Std 802.1X-2001.

4. Interlink Networks, *"Introduction to 802.1X for Wireless Local Area Networks"*, 2002.

5. Raja Azrina and Raja Othman, *"Understanding the Various Types of Denial of Service Attack"*, January 2002.

6. Chandan Singh Negi, *Using Network Management Systems to Detect Distributed Denial of Service Attacks*, September 2001.

7. Computer Network Laboratory, *"DDoS Attacks, Simulation and Detection Projects"*, February 2002.

8. T.Dierks, C. Allen *"The TLS Protocol Version 1.0"*, RFC-2246 January 1999.

9. John Bellardo and Stefan Savage, *802.11 Denial-of-Service Attacks: Real Vulnerabilities and Practical Solutions,* University of California at San Diego, USENIX Security Symposium August 2003.

10. Huseyin Selcuk Ozturk, *Evaluation of Secure 802.1X Port-Based Network Access Over 802.11 Wireless Local Area Networks*, Master Thesis March 2003.

11. Behrouz A. Forozuan, *Local Area Networks,* McGraw Hill, 2003.

12. Mishra, W. Arbaugh, *"An Initial Security Analysis of the IEEE 802.1X Standard"*, CS-TR-4328 UMIACS-TR-2002-10 Technical Report 6 February 2002.

13. Cisco Systems, Inc., *Cisco Aironet Response to University of Maryland's Paper, "An Initial Security Analysis of the IEEE 802.1x Standard",* 22 August 2002.

14. Adam Stubblefield, John Ioannidis, and Aviel D. Rubin, *"Using Fluhrer, Mantin, and Shamir Attack to Break WEP"*, AT&T Labs Technical Report 2001.

15. Cisco Systems, Inc. *"Extensible Authentication Protocol Transport Layer Security Deployment Guide for Wireless LAN Networks* ", White Paper, November 2002.

16. Ping Ding, JoAnne Holliday, Aslihan Celik, "*Improving the Security of Wireless LANs by Managing 802.1x Disassociation"*, Proceedings of th IEEE Consumer Communications and Networking Conference, Las Vegas, NV, January 2004

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Geoffrey Xie
   Naval Postgraduate School
   Monterey, CA

4. Professor John Gibson
   Naval Postgraduate School
   Monterey, CA

5. Deniz Kuvvetleri Komutanligi
   Personel Daire Baskanligi
   06410 Bakanliklar
   Ankara, TURKEY

6. Bogazici Universitesi
   Muhendislik Fakultesi, Bilgisayar Muhendisligi
   34342 Bebek
   Istanbul, TURKEY

7. Ortadogu Teknik Universitesi
   Muhendislik Fakultesi, Bilgisayar Muhendisligi
   Ankara, TURKEY

8. Orhan OZAN
   Salcikir Asfalti No:133 Tarabya 80880
   Istanbul, TURKEY